microware

OS-9/68000
C COMPILER
USER'S MANUAL

OS-9/68000
C COMPILER
USER'S MANUAL

Copyright 1984 Microware Systems Corporation, All Rights Reserved. Reproduction of this document, in part or whole, by any means, electrical or otherwise, is prohibited, except by written permission from Microware Systems Corporation.

The information contained herein is believed to be accurate as of the date of publication, however, Microware will not be liable for any damages, including indirect or consequential, from use of the OS-9 operating system or reliance on the accuracy of this documentation. The information contained herein is subject to change without notice.

OS-9 is a trademark of Microware System Corp. and Motorola Inc.
OS-9/68000 is a trademark of Microware System Corp.
UNIX is a trademark of Bell Laboratories.
VMS is the trademark of Digital Equipment Corp.

Revision C
Publication date: July, 1985
Publications Editor: Walden Miller

Microware Systems Corporation
1866 N.W. 114th Street
Des Moines, Iowa 50322
(515) - 224-1929

PMN CCC68

OS-9/68000 C COMPILER USER'S MANUAL

TABLE OF CONTENTS

Introduction i

Chapter 1 - Installing and Running the Compiler

- Executable Files 1-1
- Library Files 1-1
- System Default Device 1-2
- Command Lines and the C Executive 1-2
- Compiler Options 1-5
- Example Command Lines 1-7
- Temporary Files and Optimizing Compilation Speed 1-8

Chapter 2 - The Compiler Implementation

- Data Representation 2-1
- Register Variables 2-1
- Floating Point Data Formats 2-2
  - Single Precision Storage Format 2-2
  - Double Precision Storage Format 2-2
- Access to Command Line Parameters (Argc, Argv, Envp) 2-3
- End-of-Line Character 2-3
- Implementation-Dependent Variances 2-3
- Enhancements and Extensions 2-4
  - Imbedded Assembly Language 2-4
  - The Remote Storage Class 2-4
  - Control Character Escape Sequences 2-6
- System Calls and the Standard Library 2-7
  - Operating System Calls 2-7
  - The Standard Library 2-7

Chapter 3 - Compiler Organization

- Compiler Organization 3-1
  - Object Code Output 3-1
  - The "Cstart" Routine 3-2
  - Run-Time Arithmetic Error Handling 3-2
  - The Stack 3-3
  - Interfacing To Assembly Language 3-4

Chapter 4 - The Standard Library

- The C Standard Library 4-1
  - Standard I/O Functions 4-2
  - Character Classification Functions 4-2
  - Character Conversion Functions 4-2
  - String Handling Functions 4-2
  - Mathematical Functions 4-3
  - Memory Management Functions 4-3
  - Miscellaneous Functions 4-4
  - OS-9 System Functions 4-4
  - UNIX-like System Functions 4-4
- Function Descriptions 4-6

TABLE OF CONTENTS

Appendix A - Error Messages A-1

Appendix B - Command Lines / Options B-1

Appendix C - Function Index C-1

Index

OS-9/68000 C COMPILER USER'S MANUAL

INTRODUCTION

Microware's 68000 C Language Compiler System is an advanced technology, high-performance software development tool with the following features:

- Comprehensive implementation of the full language
- Extremely efficient code generation producing extremely fast and compact object programs
- Generates position-independent, reentrant, ROMable code
- High compilation speed
- UNIX[*] and OS-9 compatible standard libraries.

The 68000 microprocessor, the OS-9 operating system and the C language form an outstanding combination. The 68000's stack-oriented instruction set and versatile repertoire of addressing modes handle the C language very well. The OS-9 C compiler was designed specifically for the 68000 and takes full advantage of these abilities and features.

This compiler also serves as a gateway between UNIX and OS-9. Because of the compiler compatibility and similarities of OS-9 and UNIX, almost any application program written in C can be transported fr[illegible] a UNIX system to an OS-9 system, recompiled and correctly executed. The compiler can also be run on UNIX-based computers and the output downloaded to an OS-9-based 68000 system.

Cross Compiler Versions

The OS-9/68000 C compiler is available in cross compiler versions for use on various development systems. The systems currently supported are listed below:

Host System	Operating System
DEC VAX Series	UNIX System V, BSD 4.2, VAX/VMS

Unless otherwise noted, the information given in this document applies to native mode and cross versions of the compiler.

INTRODUCTION

"The C Programming Language" - Kernighan & Ritchie

The specification for the OS-9/68000 C compiler is the book "The C Programming Language" by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice Hall, Inc., and hereafter referred to as "K&R". This compiler conforms exactly to this specification except for the implementation dependent characteristics noted in this manual.

K&R frequently refer to characteristics of the C language whose exact operations depend on the architecture and instruction set of the computer actually used. This manual contains specific information regarding this version of C for the 68000 processor.

ACKNOWLEDGMENTS

The Microware 68000 C compiler and the Microware 68000 Macro Assembler and Linker were written by Kim Kempf and Larry Crane. Special thanks is due James McCosh who wrote the original C compiler for the 6809 upon which this compiler is based. Thanks also to Robert Doggett, Bill Phelps, Doug Nicholson, Todd Earles, Wes Camden, Eric Miller, Rick Grewell, Steve Guntly and Ken Kaplan for their patience, constructive criticisms and testing.

OS-9/68000 C COMPILER USER'S MANUAL
CHAPTER 1
INSTALLING AND RUNNING THE COMPILER

Executable Files

The following binary files comprise the executable part of the compiler system. These files should be copied from the distribution media to the "CMDS" directory on OS-9 systems:

cc	compiler executive program ("cc68" for cross versions)
cpp	macro preprocessor
c68	compiler pass
o68	optimizer
r68	OS-9/68000 macro assembler
l68	OS-9/68000 linker
cio	optional trap handler for the C I/O Library

Library Files

The following are important files used during the compilation process. These files must be located in a directory named "LIB" on the system's default mass storage device.

cstart.r	contains startup code for compiled programs.
clib.l	contains the standard library, math functions and the system library.
clibn.l	contains the same as clib.l but does not use the math1 trap handler for floating point support. The routines to perform floating point are extracted from the math.l library.
math.l	contains the same floating point functions as the math trap handler, but in stand alone form.
sys.l	contains the system global definitions.

If angle brackets ("<" and ">") are used instead of parentheses when specifying "#include" files, the files will be searched for starting at the "DEFS" directory on the default system device.

System Default Device

The C compiler system relies on a number of header ("#include") and library files to compile a program. These files are kept in the directories named "DEFS" and "LIB", respectively.

The C executive uses a heuristic approach to determine the "system default device". "cc" will try to open, in this order, the devices "/dd", "/h0" and "/d0". The first device that can be opened is assumed to contain the DEFS and LIB directories.

Since the devices are not searched, the directories should appear on the first device in the search list that is present on the system. This method is an attempt to locate the directories on the fastest device possible using the Microware naming conventions: /dd ramdisk, /h0 = hard disk and /d0 = floppy disk.

The Shell environmental variable, DDEV, can be used to specify the default device. If DDEV is defined, the specified device is assumed to be the default device. It is further assumed to contain the DEFS and LIB directories.

To use DDEV to define the default device, enter a Shell command using the following syntax:

```
setenv DDEV <device name>
```

To define the default device as /h0, you would enter:

```
setenv DDEV /h0
```

DDEV will stay defined with the specified device name for the life of the Shell.

Command Lines and the C Executive

The compiler system is managed for you by a small C executive program called "cc" (or "cc68" in cross-compiler versions). The executive accepts a single simplified command line and automatically calls the other parts of the compiler system as needed.

The syntax of the command line which calls the compiler is:

```
cc [ option-flags ] file {file} [ option-flags ]
```

OS-9/68000 C COMPILER USER'S MANUAL
CHAPTER 1
INSTALLING AND RUNNING THE COMPILER

One file at a time can be compiled, or a number of files may be compiled together. You can mix C or assembler source code and relocatable type files on the command line. The C executive manages the compilation through as many as four stages: preprocessor, compilation to assembler code, assembly to relocatable module, and linking to binary executable code (in OS-9 memory module format).

The compiler accepts three types of source files, provided each name on the command line has the relevant suffix as shown below. Any of the below file types may be mixed on the command line.

Suffix	Usage
.c	C source file
.a	assembly language source file
.r	relocatable module

There are two basic modes of operation: multiple source file and single source file. The compiler selects the mode by inspecting the command line. The usual mode is single source file mode and is specified by having only one source file name on the command line.

In this mode, the compiler will use the name obtained by removing the suffix from the name supplied on the command line. The output file (and the memory module produced) will have this name. The following example will leave an executable file called "prg" in the execution directory:

```
cc prg.c
```

The multiple source file mode is specified by listing more than one source file name on the command line. In this mode, the object code output file will have the name "output" in the current execution directory, unless a name is given using the "-f=" option (see below).

Also in multiple source file mode, the relocatable modules generated as intermediate files will be left in the same directories as their corresponding source files with the suffixes changed to ".r".

The following example will leave an executable file called "output" in the current execution directory, a relocatable file called "prg1.r" in the current data directory and "prg2.r" in "/d0/fred":

```
cc prg1.c /d0/fred/prg2.c
```

The following command line compiles the file called "PNAME.C" into an executable object file named "PROG" and sets the module revision level to 3:

```
cc -E=3 PNAME.C -F=PROG
```

The following command line compiles the program with a definition identifier being passed to the compiler. The definition being passed is used within the source to control compilation via IFDEF/IFNDEF functions.

```
CC PROG.C -DFLOATS
```

Compiler Options

The compiler recognizes many command line option flags which modify the compilation process where needed. All flags are recognized before compilation commences so the flags may be placed anywhere on the command line. Flags may be grouped together ("-sr") except where a flag is followed by something else, as "-f=<path>".

cc (C Language Executive)

Options are not case significant.

OPTION	DESCRIPTION
-a	suppresses the assembly phase, leaving the output as assembler code in a file with the ".a" suffix.
-c	will output the source code as comments with the assembler code. This option is most useful with the -a option.
-d<identifier>	is equivalent to a "#define <identifier>" written in the source file. This option is useful where different versions of a program are maintained in one source file and differentiated through the "#ifdef" or "#ifndef" preprocessor directives. If the <identifier> is used as a macro for expansion by the preprocessor, "1"(one) will be the expanded "value" unless the form "-d<identifier>=<string>" is used in which case the expansion will be the specified <string>.
-e=<number>	sets the edition number constant byte to the specified number. This is an OS-9 convention for memory modules.
-f=<path>	overrides the output file naming conventions. The output file will be left with <path> as its name. This flag does not make sense in multiple source file mode when either the -a or -r flag is also present. The module will be called the last name in <path>.
-g	causes the linker to output a symbol module for use by the symbolic assembly language level debugger. The symbol module is placed in the execution directory using the name of the output module with the suffix: ".stb".

cc (C Language Executive)

OPTION	DESCRIPTION
-i	links the program with the "cio.l" library which causes references to selected C I/O functions to be handled by a trap handler module called "cio".
-l=<dir>	specifies a library file to be searched by the linker before the standard library, math libraries and system interface library.
-m=<mem size>	instructs the linker to allocate <mem size> for the program stack. Memory size is given in kilobytes. The default stack size is approximately 2k bytes.
-o	inhibits the assembly code optimizer pass. The optimizer will shorten object code by about 11% with a comparable increase in speed. It is recommended for production versions of debugged programs.
-q	(quiet mode): the executive will not announce internal steps as they occur. Only error messages, if any, are displayed.
-r[=<dir>]	suppresses linking library modules into executable programs. Output is left in files with a ".r" suffix. If -r=<dir>, then ".r" files stay in <dir>
-s	stops the generation of stack-checking code. This option should only be used with great care when the application is extremely time critical and when the use of the stack by compiler generated code is fully understood.
-t=<dir>	causes the executive to place the temporary files used by any compiler phase in the directory named <dir>. If the device containing the directory is the ramdisk device (e.g. -t=/r0), compilation time will be drastically reduced.
-u<name>	will "UN-define" previously defined pre-processor macro names. Macro names pre-defined in the pre-processor are "OSK" and "mc68000". These names are useful for identifying the compiler under which the program is being compiled for purposes of writing machine and operating system independent programs.

cc (C Language Executive)

OPTION	DESCRIPTION
-v=<dir>	specifies a directory to search for pre-processor "#include" files. The files must be specified within angle brackets (<>). The directory must contain ALL the "#include" files used. This path will be used instead of the DEFS directory on the system default device. This option may appear more than once. In this case each directory is searched in order of appearance for the specified file.
-x	causes the compiler to generate trap instructions to access the floating point math routines. This option should appear on the command line when the program is both compiled and linked (if performed separately). The linker will cause the object program to use the trap handler modules rather than extracting the code from the math libraries.

Example Command Lines

Command line	Action	Output file(s)
cc prg.c	compile to an executable program	prg
cc prg.c -a	compile to assembly language source code	prg.a
cc prg.c -r	compile to relocatable module	prg.r
cc prg1.c prg2.c prg3.c	compile to executable program	prg1.r, prg2.r, prg3.r, output
cc prg1.c prg2.a prg3.r	compile prg1.c, assemble prg2.a and combine all into an executable program	prg1.r, prg2.r, output
cc prg1.c prg2.c -a	compile to assembly language source code	prg1.a, prg2.a

Example Command Lines (continued)

Command line	Action	Output file(s)
cc prg1.c prg2.c -f=prg	compile to executable program	prg
cc sieve.c -igxf=sieve	compile to executable using the math and cio trap handlers and make a symbol module for debug.	sieve, sieve.stb

Temporary Files and Optimizing Compilation Speed

A number of temporary files are created in the current working data directory during compilation, and it is important to ensure that enough space is available on the disk drive. As a rough guide, at least three times the number of blocks in the largest source file (and its #include files) should be free.

The compilation speed is directly effected by the speed at which the temporary file can be written and read. The working directory used for compilation should be the fastest device available on the system. For example, if your OS-9 system has a memory-based virtual disk device it should be used for your working directory. In addition, the "-t" command line option can be used to specify a fast device on which to place any temporary files.

end of chapter 1

OS-9/68000 C COMPILER USER'S MANUAL
CHAPTER 2
COMPILER IMPLEMENTATION

DATA REPRESENTATION

Each variable type requires a specific amount of memory for storage. The sizes of the basic types (in bytes) are as follows:

Data Type	Bytes	Internal Representation
char	1	two's complement binary
unsigned char	1	unsigned binary
short	2	two's complement binary
unsigned short	2	unsigned binary
int	4	two's complement binary
unsigned	4	unsigned binary
long	4	two's complement binary
float	4	binary floating point (see below)
double	8	binary floating point (see below)

NOTE: The compiler follows the convention that char and short are converted to **int** with sign extension. The data types long and int are synonymous.

Register Variables

A considerable savings in code size and speed can be made by judicious use of register variables. When heavily used variables are declared as **register** type storage, the compiler can perform many optimizations on-the-fly that it could not otherwise. The most efficient use is for a pointer or a loop counter.

Register declarations can be used on automatic variables or function arguments. The **register** declaration will be effective only for integral or pointer types. Any invalid **register** types or declarations in excess of available registers are simply ignored. By declaring as many **register** variables as possible, the best possible object code is obtained.

Within each function, three 32-bit address (An) registers are available for pointers and four 32-bit data (Dn) registers are available for non-pointer variables. If a **register** declaration is inappropriate for the type (like **float** or **double**), the declaration is simply ignored (i.e., the storage class is made automatic).

The register assignments for the variables are made in the order that the declarations appear. Therefore, it is wise to declare variables as **register** in the order of heaviest usage. If more **register** declarations are given than registers available, the excess **register** storage class declarations are considered automatic.

For further details see K&R on "register declaration".

Floating Point Data Formats

The compiler uses a floating point representation based on IEEE Draft Standard 754, which has been adapted as the standard OS-9/68000 format. It is intended that use of this format will afford direct compatibility with floating point arithmetic coprocessor hardware.

Single Precision Storage Format

1	8 bits	23 Bits
sign	exponent	significand (mantissa)

MSB LSB

Range: approx. +/- 1.2*10^-38 to 3.4* 10^38
Precision: approx. 7 decimal digits.

Double Precision Storage Format

1	11 bits	52 Bits
sign	exponent	significand (mantissa)

MSB LSB

Range: approx +/- 2.2*10^-308 to 1.8*10^308
Precision: approx. 16 decimal digits

The mantissa has an implied leading "1" bit. The sign bit is the most significant bit of the value. The exponent is biased by 128 for floats and 1024 for doubles.

As specified in K&R, floats are promoted to double for actual computation, so in effect, float exists only as a storage representation; no speed increase is realized. The floating point code is normally accessed by the compiler via traps to the OS-9 floating point trap handlers, thereby saving considerable memory.

Although the compiler system uses the storage representation set forth in the specification, the actual mathematical library routines do not support certain internal details of the standard such as infinity arithmetic, NANs, user defined rounding options, etc. These functions are of use to a very limited class of users, do not affect future compatibility and if supported (in software routines) would significantly degrade execution speed.

ACCESS TO COMMAND LINE PARAMETERS (Argc, Argv, Envp)

The standard C arguments "argc" and "argv" are available to "main" as described in K&R on page 110. The "cstart.r" start-up code converts the parameter string passed to it by the parent process into null-terminated strings as expected by the K&R standard. The argument "envp" points to a list of environment variables for the process. The environment variables are normally set by the Shell. See the discussion of the "cstart.r" startup module in chapter 3 for the details on argument and environment variable handling.

END-OF-LINE CHARACTER

The escape sequence for new-line ("\n") refers to the ASCII carriage return character (used by OS-9 for end-of-line), not linefeed (hex 0A) as used in UNIX. Despite this difference, programs using "\n" for end-of-line (including all programs in K&R) will work properly.

IMPLEMENTATION-DEPENDENT VARIANCES

Although C is a very portable language, there are inevitably minor differences between versions. This compiler is no exception, and its differences mostly reflect parts of C that are either obsolete or hardware dependent. The list below describes characteristics of the compiler that may affect program portability, usually in rare cases.

1. Bit fields are not supported in the present release. They will be supported in a future release.

2. The older (obsolete) forms of the assignment operators, "=+" or '=*', are not supported. The newer forms ("+=", "*=", etc.) must be used.

3. The pre-processor directives "#ifdef...#else...#endif" or '#ifndef..." are supported, but "#if <constant expression>" is not.

4. The following UNIX version 7 features are not yet supported but will appear in a future release:

   A. Structure assignments and functions returning data of type struct.

   B. Enumerated data types.

   C. Non-unique struct member names.

ENHANCEMENTS AND EXTENSIONS

The compiler includes a number of additional useful features as listed below. These features are non-standard and may affect the portability of programs to other C compilers.

Imbedded Assembly Language

As versatile as C is, occasionally there are some things that can only be done (or done at maximum speed) in assembly language. The OS-9 C compiler permits user-supplied assembly language statements to be directly embedded in C source programs.

A line beginning with "#asm" switches the compiler into a mode which passes all subsequent lines directly to the assembly language output, until a line beginning with "#endasm" is encountered. "#endasm" switches the mode back to normal.

Alternatively, a single line beginning with an @ causes the compiler to pass the remainder of the line directly to the assembler.

Care should be exercised when using embedded assembly language so that the correct code section is adhered to. Normal code from the compiler is in the PSECT (code) section. If your assembly code uses the VSECT (variable) section, be sure to put an ENDSECT directive at the end to leave the state correct for the following compiler generated code.

The -a and -c compiler options are useful to check out the embedded assembler code.

Consult the section of this manual entitled "Compiler Organization" for complete details on assembly language C functions.

The Remote Storage Class

The C compiler supports an additional storage class: the "remote" storage class.

The 68000 "register indirect with offset" addressing mode limits the adressing range to 64K. If a program requires more than 64K of static or global storage, the "indexed register indirect with offset" addressing mode must be used. This is the only way to index an address register with a 32 bit offset.

The compiler generates variable references with 32 bit offsets for those variables declared as "remote". All "remote" data for a program is assigned by the linker after the "normal" data allocations.

The most common use for the remote storage class is to declare very large arrays:

```
remote double bigarray [10000];
```

This declaration reserves 80,000 bytes of storage for the 10,000 element "double" array. The linker will organize data memory for the process using "bigarray" as follows:

```
High Address     .--------. -.
                 |        |  |
                 |        |  |
                 |        |  |
                 |        |  |
                 |        |  >-- 80K array for bigarray[]
                 |        |  |
                 |        |  |
                 |        |  |
                 |        | _|
                 >--------< -.
Remote branch    |        |  |
jump table       |        |  |
(if required)    >--------<  |
                 |        |  >-- initialized variables
                 |        |  |
                 |        |  |
                 |        | _|
                 >--------< -.
                 |        |  |
                 |        |  >-- "normal" static and global
                 |        |  |   variables
                 |        |  |
Low Address      '--------' -' < base of data area
```

The remote storage class should be used sparingly, since the code required to access such variables is larger and slower than accessing non-remote variables.

As an alternative to using remotes, a program could declare a pointer to a lare array and perform a memory request to the system to obtain the necessary storage.

The syntax for the remote storage class is identical to that of the extern storage class:

```
remote <data type> <variable>
```

The scope of the declared variable can be limited to the current file by specifying:

```
static remote <data type> <variable>
```

Remote variables can only be declared outside of a function. Functions can not declare remote storage.

Remote variables can not be initialized.

Control Character Escape Sequences

The escape sequences for non-printing characters in character constants and strings (see K&R on "character and string constants") are extended as follows:

```
linefeed (LF):  \l (lower case 'ell')
```

This is to distinguish LF (hex 0A) from \n which on OS-9 is the same as \r (hex 0D).

```
bit patterns:  \NNN    (octal constant)
               \dNNN   (decimal constant)
               \xNN    (hexadecimal constant)
```

For example, the following all have a value of 255 (decimal):

```
\377          \xff          \d255
```

SYSTEM CALLS AND THE STANDARD LIBRARY

Operating System Calls

The system interface supports almost all of the system calls of both OS-9 and UNIX. In order to facilitate the portability of programs from UNIX, some of the calls use UNIX names rather than OS-9 names for the same function. Even though the names are the same, consult the discussion of the function for any possible differences between the UNIX system call and the OS-9 equivalent.

There are a few UNIX calls that do not have exactly equivalent OS-9 calls. In these cases, the library function simulates the function of the corresponding UNIX call. In cases where there are OS-9 calls that do not have UNIX equivalents, the OS-9 names are used. Details of the calls and a name cross-reference are provided in Chapter 4 of this manual.

The Standard Library

The C compiler includes a very complete library of standard I/O functions provided with most UNIX or UNIX-like systems. It is essential for any program which uses the high-level I/O functions from the standard library (such as "fopen", "getc", "putc", etc.) to include the statement:

```
"#include <stdio.h>"
```

See Chapter 4 of this manual for individual details on the standard library functions provided.

end of chapter 2

USER NOTES

COMPILER ORGANIZATION

The C compiler is made up of three main parts: the preprocessor (cpp), the compiler (c68), and the object code improver (o68).

The Microware 68000 Macro Assembler (r68) and linker (l68) are required to assemble and link the compiler output.

The C executive (cc or cc68) provides a uniform command interface between the user's command line and each individual compiler phase.

It is the job of the preprocessor to condition the source program for the compiler. Conditioning involves gathering and expanding macros, including the contents of other files into the output, removing comments and providing the compiler with information required to report the files and lines in error (if any).

The compiler then takes the preprocessor output and (in a single pass) translates the source lines into assembly source code suitable for the Microware 68000 Macro Assembler.

Any errors in the source program are displayed on the standard error path with the associated source line and a caret (^) pointing to the item causing the error. For example:

```
        prog.c: line 10  **** undeclared variable ****
            i = j + 1;
                ^
```

Due to the one-pass nature of the compiler, the object code improver may be invoked to "clean-up" any inefficient code left in the assembly source by the compiler. This function identifies and removes needless instruction sequences. It recognizes and compresses dynamically identical program logic, changes long displacement relative branches to short where possible and removes comment lines.

Object Code Output

The C compiler produces assembly code that is normally position independent, re-entrant and ROMable assuming good programming techniques are employed. It is possible, by the very nature of the C language, to write code that modifies itself or accesses memory by absolute address. This practice should be avoided to be consistent with the OS-9 position independent shared-module philosophy.

After the assembly code is assembled and linked, the output of the linker is in the form of a standard OS-9 executable machine language memory module (including module header and CRC check value).

The "Cstart" Routine

The C compiler system depends on a short code section called "cstart.r" as the first section of every program. It includes:

1. The assembler directives to create an executable program module.
2. Code to convert the parameter string passed by F$Fork into the argv and envp strings for main().
3. Code to initialize the C I/O facility.
4. Other initialization code.
5. A call to the function "main()".

Although the compiler is intended to produce code for OS-9-based target systems, it is possible to alter the "cstart" routine for non-OS-9 or stand alone systems. The assembler source code for this routine is provided in a file called "cstart.a". If you use the compiler to produce programs for non-OS-9 target systems you will also need to provide your own versions of the standard library functions that perform I/O, memory management, etc.

n-Time Arithmetic Error Handling

K&R leave the treatment of various arithmetic errors open, merely saying that it is machine dependent. This implementation deals with a limited number of error conditions in a special way.

Integer division by zero causes a trap through the "divide by zero" machine vector. This causes a program to exit with the error status code of E$ZerDiv (000:105) unless a handler for this exception is provided.

All floating point errors, including division by zero, cause a trap through the "TRAPV instruction" machine vector. This causes the program to exit with the error status code of E$TrapV (000:107). These and other machine exceptions may be caught with a user-supplied assembly language routine.

Results of other possible errors are undefined.

The Stack

The upper part (higher addresses) of a C program's data area is reserved for the stack. Each procedure invocation uses stack space for linkage and automatic variable space.

It is impossible for the compiler to determine how much stack space will be required by a specific program. Programs that nest function calls very deeply or are highly recursive will require more stack space. The default amount of stack allocated is approximately 2K. You may increase the stack to any desired size using the "-m" compiler option flag.

The compiler generates code to check for stack underflow during procedure calls. The compiler's "-s" option flag can be used to inhibit generation of stack checking code. Though the extra code produced makes slightly larger and slower programs, it is recommended that the stack checking code be retained until a program is well tested. Stack overflow bugs are among the most difficult to locate.

Interfacing To Assembly Language

C programs can run hand-written assembly language either by inline coding in C programs using the #ASM and #ENDASM directives or by giving the name of previously assembled relocatable file(s) on the C executive command line.

It is very difficult to determine just where the compiler is during code generation, therefore it is not recommended that assembly language code be embedded within C functions. Assembly code is best placed outside of a function declaration or in a separately assembled module to be linked in with the rest of the program.

All functions and machine language subroutines are called by BSR instructions except functions more than 32k bytes away.

The register usage conventions are (in general) as follows:

D0 - D1 = Function argument/return registers
D2 - D3 = Compiler allocated temporaries
D4 - D7 = Used for user register variables

A0 - A1 = Compiler allocated temporaries
A2 - A4 = Used for user register variables
A5 = Reserved for future use
A6 = Base address of variable storage area
A7 = Stack Pointer

The compiler uses a complex register allocation method to provide the smallest, fastest code for the majority of programs encountered. The 68000 has a large number of processor registers. About half of these are made available for use as register variables. The rest are used by the compiler for storage of intermediate results during the evaluation of expressions.

The A6 register (used as a pointer to the base of the global and static variables) is passed to a program when the program is forked and is never changed by C code.

The type of the argument and the order specified in the argument list indicates to the called function where the argument is: it will either be in a register or on the stack.

For this discussion, an integral argument is an argument of type int, a pointer, or a **char** or **short** converted to an **int**. A double argument is an argument of type **double** or a **float** converted to a **double**.

The first integral argument is passed in d0, the second integral argument (if any) in d1. A single double argument is passed in d0 and d1, the most significant half in d0, the least significant half in d1. Any remaining arguments are pushed onto the stack. If the first argument is integral and the second is a double, the integral argument is passed in d0 and the double is passed entirely on the stack. Consult the examples of this method on the following page.

If a function is to return a value, the integral (or **float**) value is returned in the d0.l register. A **double** value is returned in d0.l and d1.l.

Examples of C argument passing techniques:

```
Assume:     int i,j,k;
            double a,b,c;

            C Code                  Generated Assembler
            ------                  -------------------
            func(i);                move.l i,d0
                                    bsr func

            func(i,j);              move.l j,d1
                                    move.l i,d0
                                    bsr func
```

Examples of C argument passing techniques (continued):

```
        C Code                 Generated Assembler
        ------                 --------------------
        func(i,j,k);           move.l k,-(sp)
                               move.l j,d1
                               move.l i,d0
                               bsr func

        func(a)                movem.l a,d0/d1
                               bsr func

        func(a,b)              move.l b+4,-(sp)
                               move.l b+0,-(sp)
                               movem.l a,d0/d1
                               bsr func

        func(a,i)              move.l i,-(sp)
                               movem.l a,d0/d1
                               bsr func

        func(i,a)              move.l a+4,-(sp)
                               move.l a+0,-(sp)
                               move.l i,d0
                               bsr func
```

All functions (C or assembler) are required to restore any changed registers to the same values they contained when the function was called. The only exceptions to this are the function return register(s) and the register(s) in which the function's argument(s) are passed.

Finally, all parameters passed on the stack are 4-byte long words. Types **char** and **short** are sign extended to long words, types **unsigned char** and **unsigned short** are padded to the left with zeroes.

The 68000 BSR instruction, which is used for function calling, is limited to a +/-32K address displacement. The BSR instruction has sufficient range for all but the very largest programs. In order to permit function calls outside this range while retaining position independence of the code, the linker (automatically) will build a jump table of long addresses of function entry points outside BSR's range.

This table resides in the data area and can be accessed by the symbol "_jmptbl" defined by the linker. When coding assembly language routines, all external functions should be accessed by the word length displacement form of BSR so the linker can change the BSR to a jump if the displacement is too distant.

The "-j" option on the cc command line will cause the linker to suppress generation of the jump table.

Consult the 68000 Macro Assembler Manual for more information on the jump table.

end of chapter 3

OS-9/68000 C COMPILER USER'S MANUAL
CHAPTER 4
THE STANDARD LIBRARY

This chapter describes the operation and usage of each library function. The functions appear in alphabetical order for quick access.

The following section generally categorizes the functions available in the standard library. The remainder of the section discusses how to interpret the function synopses.

THE C STANDARD LIBRARY

The Standard Library provided with the Microware 68000 C Compiler System consists of a collection of high-level I/O, convenience and system-level functions.

The high-level I/O functions provide facilities that are normally considered part of other languages (for example, the FORMAT statement of Fortran). C technically does not have any I/O statements, relying instead on the standard library. This enhances the versatility and portability of the language.

In addition, automatic buffering of I/O paths improves the speed of file access because fewer calls to the host operating system are required.

The high-level I/O functions should not be confused with the low-level system calls with the same names (for example, "fopen()" and "open()"). The standard library functions maintain a structure (declared as FILE in <stdio.h>) for each open file. This structure holds status information for the file. A pointer (usually supplied by "fopen()") to this structure is the "identity" of the file, and it is passed to the various I/O functions. The I/O functions will make low-level system calls when appropriate.

USING A FILE POINTER IN A SYSTEM CALL, OR A PATH NUMBER IN A HIGH-LEVEL I/O CALL, is a common mistake among beginners to C and, if made, will at best crash your program, or at worst, provide unpredictable program behavior.

In addition to the C I/O functions, the standard library contains functions to perform character classification and conversion, string manipulation, memory management, mathematical functions and system related operations.

Each function description includes a synopsis and details on the usage of the function. The synopsis shows how the function and arguments would look if written as a C function definition, even if the actual function is a macro or is written in assembly language.

For example, the synopsis for "fopen" appears as follows:

```
#include <stdio.h>

FILE *fopen(name,action)
char *name, *action;
```

The synopsis indicates that the function "fopen" requires the header file <stdio.h>, returns a pointer to a structure of type FILE, and requires two arguments, both pointers to a character string. The argument names are suggestive only, and any name can be used.

Standard I/O Functions

The following functions make up the standard I/O functions of the C library. The functions accept arguments, return values, and generally behave according to the K&R (and UNIX) library functions. All use of these functions requires the inclusion of the <stdio.h> header file:

cleareof
clearerr
fclose
fdopen
feof
ferror
fflush
fgetc
fgets
fileno
fopen
fprintf
fputs
fread
freopen
fscanf
fseek
ftell
fwrite
getc
getchar
gets
getw
pffinit
pflinit
printf
putc
putchar
puts
putw
rewind
scanf
setbuf
sprintf
sscanf
ungetc

haracter Classification Functions

The character classification functions are really macros defined in <ctype.h>. Care should be exercised when using these macros to avoid macro expansion side-effects. The macros provide a machine independent method of character classification:

isalnum
isalpha
isascii
iscntrl
isdigit
islower
isprint
ispunct
isspace
isupper
isxdigit

Character Conversion Functions

The character conversion functions provide the ability to convert a string of characters to their numeric representation and to change the case of characters:

- _tolower
- _toupper
- atof
- atoi
- atol
- toascii
- tolower
- toupper

String Handling Functions

The C language does not have a character string data type. Instead, it stores strings as character arrays and the standard library provides functions to manipulate them:

- findnstr
- findstr
- index
- rindex
- strcat
- strcmp
- strcpy
- strhcpy
- strlen
- strncat
- strncmp
- strncpy

Mathematical Functions

Transcendental and algebraic math functions provided in the Microware standard library are actually hooks into the OS-9 math trap handlers. If, for example, the math handler were to be replaced with a handler that accessed floating point hardware, the application program would require absolutely no changes to use the new trap handler. The math functions included in the standard library are those found on most UNIX systems:

- abs
- acos
- asin
- atan
- ceil
- cos
- exp
- fabs
- floor
- hypot
- log
- log10
- modf
- pow
- sin
- sqrt
- tan

Memory Management Functions

The standard library provides for a suite of functions for requesting and freeing memory in a machine and operating system independent manner. Various flavors of UNIX contain different functions, but generally provide the same results: The memory management functions provided are:

- calloc
- free
- malloc
- sbrk

For special use with OS-9, some additional functions are available for increased efficiency:

_srqmem
_srtmem
ebrk
ibrk

Miscellaneous Functions

Several miscellaneous functions appear in the standard library that ease the programmer from routine tasks:

_errmsg
_prgname
_strass
closedir
freemem
longjmp
mktemp
opendir
qsort
readdir
rewinddir
seekdir
setjmp
stacksiz
system
telldir

OS-9 System Functions

The standard library provides a large number of functions to directly access OS-9 system calls. The following functions are available to provide access to selected OS-9 system calls:

_cmpnam
_gs_devn
_gs_eof
_gs_gfd
_gs_opt
_gs_pos
_gs_rdy
_gs_size
_julian
_ss_attr
_ss_lock
_ss_opt
_ss_pfd
_ss_rel
_ss_rest
_ss_size
_ss_ssig
_ss_tiks
_ss_wtrk
_sysdate
attach
chain
chainc
chxdir
crc
create
detach
getstat
intercept
modlink
modload
munlink
munload
os9exec
os9fork
os9forkc
readln
setstat
tsleep
unlinkx
writeln

UNIX-like System Functions

Because many C programs are written for the UNIX operating system, numerous UNIX functions are used in those programs. To help provide portability for those programs, many of those functions are supported by this compiler. Most of them have very similar equivalent system calls in OS-9, others must be emulated. Some UNIX system calls can not be conveniently emulated and do not appear in the library.

Even though the system call has the same name as the UNIX equivalent and performs essentially the same operation, check the discussion of the function for any subtle differences (like the mode values on open()).

The following are the UNIX-like system calls appearing in the standard library. The function descriptions for these functions will categorize them as being "UNIX System" functions:

- _exit
- access
- chdir
- chmod
- chown
- close
- creat
- dup
- exit
- getenv
- getime
- getpid
- getuid
- kill
- lseek
- mknod
- open
- pause
- prerr
- read
- setime
- setpr
- setuid
- sleep
- unlink
- wait
- write

_cmpnam() **Compare Two Names** **_cmpnam()**

OS-9 System

SYNOPSIS:
```
int _cmpnam(target,pattern,patlen)
char *target,*pattern;
int patlen;
```

DESCRIPTION: This function performs a name comparison using the OS-9 system call F$CmpNam. _cmpnam() compares the target string to the pattern string to determine if they are the same. The target name must be terminated by a null byte. Upper and lower case are considered to match.

Two metacharacters are recognized in the pattern string: "?" matches any single character and "*" which matches any string of characters.

_cmpnam() returns 0 if the strings match. -1 is returned if no match occurs.

SEE ALSO: The F$CmpNam system request in the "OS-9/68000 Operating System Technical Manual".

_errmsg() Print an Error Message _errmsg()

Miscellaneous

SYNOPSIS:

```
int _errmsg(nerr,msg[,arg1,arg2,arg3])
int nerr;
char *msg;
```

DESCRIPTION: This function displays an error message on standard output along with the name of the program. The message string "msg" is displayed in the following format:

```
prog: <message text>
```

NOTE: "Prog" is the module name of the program and "<message text>" is the string passed in "msg".

For added flexibility in message printing, the "msg" string can be a conversion string suitable for fprintf() with up to 3 additional arguments of any type. The argument "nerr" is returned as the value of the function so _errmsg() can be used as a parameter to a function such as exit() or prerr().

EXAMPLE: Assume the program calling the function is named "foobar":

```
Call:    _errmsg(1,"programmed message\n");
Prints:  foobar: programmed message

Call:    exit(_errmsg(errno,"unknown option '%c'\n",'q'));
Prints:  foobar: unknown option 'q'
Then exits with error code in errno.
```

SEE ALSO: fprintf(), _prgname().

_exit() **Task Termination** **_exit()**

UNIX System

SYNOPSIS:
```
_exit(status)
int status;
```

DESCRIPTION: This function causes immediate termination of a program. The argument "status" provides an indication to the parent process as to the success or failure of the program.

An exit status of zero is considered normal termination; a non-zero value is interpreted as an error code by most programs (especially the shell).

Another form of this call, exit(), flushes I/O buffers and basically cleans up after the program before exiting.

This function never returns.

SEE ALSO: System call F$Exit in the "OS-9/68000 Operating System Technical Manual" _gs_devn() Get Device Name _gs_devn()

OS-9 System

SYNOPSIS:
```
int _gs_devn(path,buffer)
int path;
char *buffer;
```

DESCRIPTION: The name of the device open on a path can be determined by this function. The argument "path" is a path number of an open path and "buffer" is a pointer to a character array into which the null-terminated device name is placed. If the path number is invalid, the function returns -1 as its value and the appropriate error code is placed in the global variable "errno".

CAVEATS: Be sure to reserve at least 32 characters to receive the device name.

SEE ALSO: The I$GetStt system request, function code SS_DevNm in the "OS-9/68000 Operating System Technical Manual".

_gs_eof() Check for End-of-File _gs_eof()

OS-9 System

SYNOPSIS:

```
int _gs_eof(path)
int path;
```

DESCRIPTION: The **_gs_eof()** function determines if the file open on "path" is at end-of-file. If it is at end-of-file, the value 1 is returned. If it is not at end-of-file, 0 is returned. If "path" is invalid, -1 is returned and the appropriate error code is placed in the global variable "errno".

CAVEATS: **_gs_eof()** cannot determine end-of-file on SCF devices. SCF devices return an E$EOF error when the EOF character is read. DO NOT use this call if using the buffered I/O facility on the path. Instead, use the function **feof()**.

SEE ALSO: System call I$GetStt, function code SS_EOF in the "OS-9/68000 Operating System Technical Manual". The discussion of **feof()**.

_gs_gfd() Get File Descriptor Sector _gs_gfd()

OS-9 System

SYNOPSIS:

```
#include <direct.h>

int _gs_gfd(path,buffer,count)
int path;
struct fildes *buffer;
int count;
```

DESCRIPTION: This function will place a copy of the RBF file descriptor sector of the file open on "path" into the buffer pointed to by "buffer". A maximum of "count" bytes are copied. The structure "fildes" declared in <direct.h> provides a convenient means to access the file descriptor information.

If an error occurs, the function returns the value -1 and the appropriate error value is placed in the global variable "errno".

CAVEATS: Be sure the buffer is large enough to hold "count" bytes. This call is effective only on RBF devices. Declaring the buffer as type "struct fildes" is perfectly safe as this structure is predefined to be large enough for the file descriptor.

SEE ALSO: The I$GetStt system request, function code SS_FD in the "OS-9/68000 Operating System Technical Manual". The discussion of _ss_pfd().

_gs_opt() Get Path Options _gs_opt()

OS-9 System

SYNOPSIS:

```
#include <sgstat.h>

_gs_opt(path,buffer)
int path;
struct sgbuf *buffer;
```

DESCRIPTION: This function copies the options section of the path descriptor open on "path" into the buffer pointed to by "buffer". The structure "sgbuf" declared in <sgstat.h> provides a convenient means to access the individual option values. If the path was invalid, _gs_opt() returns -1 and the appropriate error code is placed in the variable "errno".

CAVEATS: Be sure the buffer is large enough to hold the options. Declaring the buffer as type "struct sgbuf" is perfectly safe as this structure is predefined to be large enough for the options.

SEE ALSO: The I$GetStt system request, function code SS_Opt in the "OS-9/68000 Operating System Technical Manual". The discussion of _ss_opt().

_gs_pos() Get Current File Position _gs_pos()

OS-9 System

SYNOPSIS:
```
int _gs_pos(path)
int path;
```

DESCRIPTION: This function returns the value of the file pointer for the file open on "path". If the path is invalid or the device is not an RBF device, -1 is returned as the function value and the appropriate error code is placed in the global variable "errno".

CAVEATS: This call is effective only on RBF devices. This call is unique to OS-9, the equivalent portable call is lseek(). DO NOT use this call if buffered I/O is being performed on the path, instead use ftell().

SEE ALSO: The I$GetStt system request, function code SS_Pos in the "OS-9/68000 Operating System Technical Manual". The discussion of lseek().

Test for Data Available

OS-9 System

SYNOPSIS:
```
int _gs_rdy(path)
int path;
```

DESCRIPTION: This function checks the SCF device open on "path" to see if data is waiting to be read. Read requests to OS-9 will wait until enough bytes have been read to satisfy the bytecount given thereby suspending the program until the read is satisfied.

A program can use this function to determine the number of bytes, if any, waiting to be read, and then issue a read request for only the number of bytes actually received.

If the path is invalid, no data is available to be read, or the device is not an SCF device, a value of -1 is returned as the function value and the appropriate error code is placed in the global variable "errno". Otherwise, the number of bytes available to be read is returned.

CAVEATS: This call is effective only on SCF devices. This function is not intended for use with the buffered I/O calls (like getc); unpredictable results will likely occur. Stick to low-level functions when using _gs_rdy().

SEE ALSO: The I$GetStt system request, function code SS_Ready in the "OS-9/68000 Operating System Technical Manual". The discussion of the functions read(), readln().

_gs_size() Get Current File Size _gs_size()

OS-9 System

SYNOPSIS:
```
int _gs_size(path)
int path;
```

DESCRIPTION: The function _gs_size() is used to determine the current size of the file open on "path". If the path is invalid or the device is not an RBF device, a value of -1 is returned as the function value and the appropriate error code is placed in the global variable "errno". Otherwise the size of the file is returned.

CAVEATS: This call is effective only on RBF devices.

SEE ALSO: The I$GetStt system request, function code SS_Size in the "OS-9/68000 Operating System Technical Manual".

_julian()   Convert Date/Time to Julian Value   _julian()

OS-9 System

SYNOPSIS:

```
int _julian(time,date,day)
int *time,*date;
short *day;
```

DESCRIPTION: This function takes the time and date (in standard OS-9 format) and converts them to the julian equivalents and day of week. Note that the arguments "time", "date", and "day" are pointers to the values. The "time" and "date" arguments are assumed to be in the following format:

```
        .-------.-------------.---------.----------.
time:   |   0   | hour (0-23) |  minute |  second  |
        '-------'-------------'---------'----------'
          Byte 3    Byte 2      Byte 1     Byte 0

        .---------------------.---------.----------.
date:   |   Year (2-bytes)    |  month  |   day    |
        '---------------------'---------'----------'
             Bytes 3 and 2      Byte 1     Byte 0
```

_julian() modifies the objects pointed to by its arguments as follows:

```
        .------------------------------------------.
time:   | Seconds since Midnight  ( 0 - 86399)     |
        '------------------------------------------'

        .------------------------------------------.
date:   |            Julian Day Number             |
        '------------------------------------------'

        .--------------------.
day:    |    Day of Week     |    0=Sunday, 1=Saturday
        '--------------------'
```

If an error occurs, a value of -1 is returned as the function value and the appropriate error code is placed in the global variable "errno". An example appears on the next page.

_julian() (continued)

EXAMPLE:

```
main()
{
   int date,time,tick;
   short day;

   _sysdate(&time,&date,&day,&tick);
   _julian(&time,&date,&day);

   printf("The Julian date is %d. \
            Cinderella  dies  in  %d seconds!\n",date,86400-
time);
}
```

CAVEATS: Unless debugging is an obsession, be sure to pass pointers to the date, time, and day values. Also, be sure the "day" is declared to be a short or the value will appear as day + 65536!

SEE ALSO: The F$Time and F$Julian system requests in the "OS-9/68000 Operating System Technical Manual" and the discussion of the _sysdate() function.

_prgname() Get Module Name _prgname()

Miscellaneous

SYNOPSIS: `char *_prgname()`

DESCRIPTION: This function returns a pointer to the name of the module being executed. Normally, argv[0] points to the same string, but when argv is not available, this function serves the purpose well.

CAVEATS: If the code that calls this function is executing in a trap handler, the name of the trap handler module is returned.

EE ALSO: _errmsg().

_srqmem() System Memory Request _srqmem()

Memory Management

SYNOPSIS:

```
char *_srqmem(size)
unsigned size;
```

DESCRIPTION: When tight control over memory allocation is required, _srqmem() and the complementary function _srtmem() are provided to request and return system memory.

This function is a direct hook into the OS-9 F$SRqMem system call. The specified size will be rounded to a system-defined block size. A size of 0xffffffff will obtain the largest contiguous block of free memory in the system. The global unsigned variable "_srqsiz" may be examined to determine the actual size of the block allocated.

If successful, a pointer to the memory granted is returned. If the request was not granted, the function returns the value -1 and the appropriate error code is left in the global variable "errno".

The pointer returned will always begin on an even byte boundary. Care should be taken to preserve the value of the pointer if the memory is to be returned via the _srtmem() function.

CAVEATS: This function will work only on Level One systems. The F$SrqMem request is actually intended for system level use but the extended addressing range of the 68000 required some method to obtain memory without regard to where the memory is physically located.

A user process may have up to 16 F$SrqMem requests active at a given time. Ideally, the requests should be as large as practical, and preferably some multiple of 1K bytes.

SEE ALSO: sbrk(), ibrk(), ebrk(), _srtmem(), malloc(), free().

_srtmem()   System Memory Return   _srtmem(

Memory Management

SYNOPSIS:

```
int _srtmem(size,ptr)
unsigned size;
char *ptr;
```

DESCRIPTION: This function is a direct hook into the OS-9 F$SRtMe system call. It is used to return memory granted by the _srqmem(function. Care should be taken to ensure that the "size" and "ptr are the same as returned those by _srqmem(). If an error occurs the function returns the value -1 and the appropriate error code i placed in the global variable "errno".

CAVEATS: This function will work only on Level One systems.

SEE ALSO: sbrk(), ibrk(), ebrk(), _srqmem(), malloc(), free().

_ss_attr() Set File Attributes _ss_attr()

OS-9 System

SYNOPSIS:
```
#include <modes.h>

int _ss_attr(path,attr)
int path;
short attr;
```

DESCRIPTION: A disk file's attributes can be changed with this function. The current attributes of a file can be determined with the _gs_gfd() function. The attributes of a file can be changed only by the owner of the file or the super user.

The attributes as selected in the word "attr" are set in the file open on "path". The header file <modes.h> defines the valid mode values.

If an error occurs, _ss_attr() returns -1 as its value and the appropriate error code is placed in the global variable "errno".

CAVEATS: This function is effective even if the owner or super user does not have write permission to the path. It is not permitted to set the "directory" bit of a non-directory file, or to clear the "directory" bit of a directory that is not empty.

SEE ALSO: System call I$SetStt, function code SS_Attr in the "OS-9/68000 Operating System Technical Manual". The functions _gs_gfd(), _[illegible]_pfd().

_ss_lock() **Lock Out a Record** **_ss_lock()**

OS-9 System

SYNOPSIS:

```
int _ss_lock(path,locksize)
int path;
unsigned locksize;
```

DESCRIPTION: _ss_lock() locks out a section of the file open on "path" from the current file position up to the number of bytes specified by "locksize".

If locksize is zero, all locks (record-lock, EOF lock, and file lock) are removed. If a locksize of 0xffffffff is requested, the entire file is locked out regardless of where the file pointer is. This is a special type of file lock that remains in effect until released by _ss_lock(path,0), a read or write of zero (or more) bytes, or the file is closed.

There is no way to gain a "file lock" with only read or write system calls.

If an error occurs, _ss_lock() returns -1 as its value and the appropriate error code is placed in the global variable "errno".

SEE ALSO: The I$SetStt system request, function code SS_Lock and the discussion of RBF record-locking in the "OS-9/68000 Operating System Technical Manual".

_ss_opt() Set Path Options _ss_opt()

OS-9 System

SYNOPSIS:

```
#include <sgstat.h>

_ss_opt(path,buffer)
int path;
struct sgbuf *buffer;
```

DESCRIPTION: This function copies the buffer pointed to by "buffer" into the options section of the path descriptor open on "path".

Generally, a program will fetch the options with the _gs_opt() function, change the desired values, and update the path options with the _ss_opt() function. The structure "sgbuf" declared in <sgstat.h> provides a convenient means to access the individual option values.

If the path was invalid, _ss_opt() returns -1 and the appropriate error code is left in the variable "errno".

CAVEATS: It is common practice to preserve a copy of the original options so the program can restore them prior to exiting. The option changes take effect on the currently open path (and any paths created with I$Dup to the same).

SEE ALSO: The I$SetStt system request, function code SS_Opt in the "OS-9/68000 Operating System Technical Manual". The discussion of _ss_opt() and the "tmode" utility described in the "OS-9/68000 Operating System User's Manual".

_ss_pfd() **Put File Descriptor Sector** **_ss_pfd()**

OS-9 System

SYNOPSIS:

```
#include <direct.h>

int _ss_pfd(path,buffer)
int path;
struct fildes *buffer;
```

DESCRIPTION: _ss_pfd() will copy certain bytes from the buffer pointed to by "buffer" into the file descriptor sector of the file open on "path". The buffer is usually obtained from the _gs_gfd() function. Only the owner id, the modification date, and creation date are changed.

The structure "fildes" declared in <direct.h> provides a convenient means to access the file descriptor information.

If an error occurs, the function returns the value -1 and the appropriate error value is placed in the global variable "errno".

CAVEATS: The buffer must be at least 16 bytes long or garbage could be written into the file descriptor sector. The owner ID field can be changed only by the super user. It is impossible to change the file attributes with this call. Instead, use the _ss_attr() function.

SEE ALSO: System call I$SetStt, function code SS_FD and the discussion of the RBF File Manager in the "OS-9/68000 Operating System Technical Manual". The discussion of _gs_pfd().

_ss_rel() Release Device _ss_rel()

OS-9 System

SYNOPSIS:
```
int _ss_rel(path)
int path;
```

DESCRIPTION: The function _ss_rel() cancels the signal to be sent from a device on data ready. The function _ss_ssig() enables a device to send a signal to a process when data is available. If an error occurs, the function returns the value -1 and the appropriate error value is placed in the global variable "errno".

CAVEATS: The signal request is also cancelled when the issuing process dies or closes the path to the device. This feature exists only on SCF devices.

SEE ALSO: System call I$SetStt, function code SS_SSig in the "OS-9/68000 Operating System Technical Manual". The discussion of _ss_ssig().

_ss_rest() **Restore Device** **_ss_rest(**

OS-9 System

SYNOPSIS: `int _ss_rest(path)`
`int path;`

DESCRIPTION: This function causes an RBF device to restore the disk head to track zero and is usually used for disk formatting and error recovery. If an error occurs, the function returns the value -1 and the appropriate error value is placed in the global variable "errno".

SEE ALSO: System call I$SetStt, function code SS_Reset in the "OS-9/68000 Operating System Technical Manual".

_ss_size() Set Current File Size _ss_size()

OS-9 System

SYNOPSIS: int _ss_size(path~~)~~, size)
int path; size;

DESCRIPTION: This function is used to change the size of the file open on "path". The size change is immediate.

If the size of the file is decreased, the freed sectors are returned to the system. If the size is increased, sectors are added to the file with the contents of those sectors being undefined.

If the path is invalid or the device is not an RBF device, a value of -1 is returned as the function value and the appropriate error code is placed in the global variable "errno".

CAVEATS: This call is effective only on RBF devices.

SEE ALSO: The I$SetStt system request, function code SS_Size in the "OS-9/68000 Operating System Technical Manual".

_ss_ssig() Send Signal on Data Ready _ss_ssig()

OS-9 System

```
SYNOPSIS:  _ss_ssig(path,sigcode)
           int path;
           short sigcode;
```

DESCRIPTION: The function _ss_ssig() sets up a signal to be sent to the calling process when an interactive device has data ready. When data is received on the device indicated by "path", the signal "sigcode" is sent to the calling process.

_ss_ssig() must be called each time the signal is sent if it is to be used again.

The device is considered busy, and will return an error, if any read request arrives before the signal is sent. Write requests are allowed to the device while in this state.

If an error occurs, the function returns the value -1 and the appropriate error value is placed in the global variable "errno".

CAVEATS: This feature exists only on SCF devices.

SEE ALSO: System call I$SetStt, function code SS_SSig in the "OS-9/68000 Operating System Technical Manual". The discussion of _ss_rel().

_ss_tiks() Wait for Record Release _ss_tiks()

OS-9 System

SYNOPSIS:

```
int _ss_tiks(path,tickcnt)
int path;
int tickcnt;
```

DESCRIPTION: If a read or write request is issued for a part of a file that is locked out by another user, RBF normally sleeps indefinitely until the conflict is removed. The _ss_tiks() function may be used to cause an error (E$Lock) to be returned to the program if the conflict still exists after a specified number of ticks have elapsed.

The argument "tickcnt" specifies the number of ticks to wait if a record-lock conflict occurs with the file open on "path". A tick count of zero (RBF's default) causes a sleep until the record is released. A tick count of one means that if the record is not released immediately an error is to be returned.

If an error occurs, the _ss_sig() function returns the value -1 and the appropriate error value is placed in the global variable "errno".

CAVEATS: This feature exists only on RBF devices.

SEE ALSO: System call I$SetStt, function code SS_Ticks, the discussion of RBF record-locking in the "OS-9/68000 Operating System Technical Manual" and the discussion of _ss_rel().

_ss_wtrk() Write Track _ss_wtrk(

OS-9 System

SYNOPSIS:

```
int _ss_wtrk(path,trkno,siden,ilvf,trkbuf,ilvptr)
int path;
char *trkbuf,*ilvptr;
int trkno,siden,ilvf;
```

DESCRIPTION: This function performs a write-track operation on a disk drive. It is essentially a direct hook into the driver's write-track entry point.

The argument "path" is the path on which the device is open. "trkno" is the desired track number to write. "siden" is the desired side of the track on which to write. "ilvf" is the interleave factor. "trkbuf" is the track buffer image. "ilvptr" is a pointer to an interleave table.

If an error occurs, the _ss_wtrk() function returns the value -1 and the appropriate error value is placed in the global variable "errno".

CAVEATS: This feature exists only on RBF devices. Any further information on actual use of this call can be obtained by examining the "format" utility and/or a device driver.

SEE ALSO: System call I$SetStt, function code SS_WTrk, the "format" utility or any RBF device driver.

_strass() Structure Assignment _strass()

Miscellaneous

SYNOPSIS:

```
_strass(to,from,count)
char *to,*from;
int count;
```

DESCRIPTION: Until the compiler can deal with structure assignment, this function is useful for copying one structure to another. The variable "count" specifies the number of bytes to be copied from memory pointed to by "from" to memory pointed to by "to", regardless of contents.

CAVEATS: This function can move at most 65536 bytes. No regard is given to overlapping moves.

_sysdate() Get Current System Date/Time _sysdate()

OS-9 System

SYNOPSIS:

```
int _sysdate(format,time,date,day,tick)
int format,*time,*date,*tick;
short *day;
```

DESCRIPTION: This function obtains the current time, date, day of week, and clock tick from the system. Note that all the arguments except "format" are pointers to the receiving locations.

"format" can be any of the following:

```
0 = Gregorian              2 = Gregorian with ticks
1 = Julian                 3 = Julian with ticks
```

The values are returned in the following format:

```
         .-------.------------.--------.----------.
time:    |   0   | hour (0-23) | minute |  second  |   (Gregorian)
         '-------'------------'--------'----------'
          Byte 3     Byte 2     Byte 1     Byte 0

         .----------------------------------------.
         | Seconds since midnight (long) (0-86399) |   (Julian)
         '----------------------------------------'

         .-------------------.--------.----------.
date:    |  Year (2-bytes)   | month  |   day    |   (Gregorian)
         '-------------------'--------'----------'
           Bytes 3 and 2      Byte 1     Byte 0

         .----------------------------------------.
         | Julian day number (long)               |   (Julian)
         '----------------------------------------'

         .-------------------.
day:     |    Day of Week    |     0=Sunday, 6=Saturday
         '-------------------'

         .-------------------.--------------------.
tick:    | No. of Ticks/Second | Current Clock Tick |
         '-------------------'--------------------'
```

_sysdate() (continued)

If an error occurs, a value of -1 is returned as the function value and the appropriate error code is placed in the global variable "errno".

EXAMPLE:

```
main()
{
   int date,time,tick;
   short day;

   .
   .
   .

   _sysdate(0,&time,&date,&day,&tick);

   .
   .
   .
}
```

CAVEATS: Unless debugging is an obsession, be sure to pass pointers to the date, time, and day values. Also, be sure the "day" is declared to be a short or the value will appear as day + 65536!

SEE ALSO: System call F$Time, F$Julian in the "OS-9/68000 Operating System Technical Manual" and the discussion of _julian().

_tolower() Convert Character to Lower Case _tolower()

Character Conversion

SYNOPSIS:

```
#include <ctype.h>

int _tolower(c)
char c;
```

DESCRIPTION: _tolower() is a macro that changes the lowercase argument to uppercase. The argument MUST be lowercase or garbage will result. The function tolower() should be used if the argument is not guaranteed to be lowercase.

SEE ALSO: toupper(), isascii().

_toupper()   Convert Character to Upper Case   _toupper()

Character Conversion

SYNOPSIS:

```
#include <ctype.h>

int _toupper(c)
char c;
```

DESCRIPTION: _toupper() is a macro that changes the lowercase argument to uppercase. The argument MUST be lowercase or garbage will result. The function toupper() should be used if the argument is not guaranteed to be uppercase.

SEE ALSO: tolower(), isascii().

abs() Integer Absolute Value **abs()**

Mathematical

SYNOPSIS:

```
int abs(value)
int value;
```

DESCRIPTION: abs() returns the absolute value of its integer argument.

CAVEATS: Applying abs() to the most negative integer yields a result which is the most negative integer:

abs(0x80000000) returns 0x80000000 as the result.

access() Determine Accessibility of a File **access()**

UNIX System

SYNOPSIS:

```
#include <modes.h>

int access(name,perm);
char *name;
short perm;
```

DESCRIPTION: The function access() returns 0 if the mode(s) specified in "perm" are correct for user to access "name".

The value for "perm" may be any legal OS-9 mode as defined in the header file <modes.h>. A mode value of zero can be used to verify the existence of a file.

If the file can not be accessed, the function returns -1 and the appropriate error code is placed in the global variable "errno".

CAVEATS: Note that the "perm" value may not be compatible with other systems.

acos() **Arc Cosine** **acos(**

Mathematical

SYNOPSIS:

```
#include <math.h>

double acos(x)
double x;
```

DESCRIPTION: acos() returns the arccosine of "x", in the range of to PI.

asin() Arc Sine asin()

Mathematical

SYNOPSIS: #include <math.h>

```
double asin(x)
double x;
```

DESCRIPTION: asin() returns the arc sin of x, in the range -PI/2 to PI/2.

atan() **Arc Tangent** **atan()**

Mathematical

SYNOPSIS:

```
#include <math.h>

double atan(x)
double x;
```

DESCRIPTION: atan() returns the arc tangent of x, in the range -PI/2 to PI/2.

atof() Alpha to Floating Conversion atof()

Character Conversion

SYNOPSIS:

```
#include <math.h>

double atof(string)
char *string;
```

DESCRIPTION: The function atof() converts the string pointed to by "string" into its equivalent representation in type double.

atof() recognizes the following syntax: an optional sign followed by a digit string (possibly containing a decimal point), an optional "e" or "E", an optional sign, and a digit string (no decimal point):

```
[+/-]digits[.digits] [E/e[+/-]integer]
```

CAVEATS: Overflow causes unpredictable results. There are no error indications.

atoi() **Alpha to Integer Conversion** atoi()

Character Conversion

SYNOPSIS: int atoi(string)
char *string;

DESCRIPTION: The function atoi() converts the string pointed to by "string" into its equivalent representation in type int. atoi() recognizes an optional sign followed by a digit string:

```
[+/-]digits
```

CAVEATS: atoi() and atol() are identical functions in this implementation.

atol() **Alpha to Long Conversion** **atol()**

Character Conversion

SYNOPSIS:
```
long atol(string)
char *string;
```

DESCRIPTION: This compiler implementation treats long and int values identically. This function is identical to atoi(), and is included here for portability considerations. See the discussion of atoi() for operational details.

attach() **Attach to a Device** **attach()**

OS-9 System

SYNOPSIS:

```
#include <modes.h>

char *attach(name,mode)
char *name;
short mode;
```

DESCRIPTION: The attach() function is useful to cause a device to become "known" by the system, or verify that it is already attached.

If the device is not already attached, the device is placed in the system's device table, static storage is assigned to the device, and its initialization routine is executed. If the device is already attached, it will not be re-initialized.

The device name passed in the argument "name" is attached with the access mode passed in the argument "mode". If the attach is successful, the device table address is returned as the value of the function. If the attach fails, -1 is returned and the appropriate error code is placed in the global variable "errno".

SEE ALSO: The I$Attach system call in the "OS-9/68000 Operating System Technical Manual".

calloc() **Allocate Storage for Array** **calloc()**

Memory Management

SYNOPSIS: char *calloc(nel,elsize)
unsigned nel,elsize;

DESCRIPTION: This function allocates space for an array. The argument "nel" is the number of elements in the array, and "elsize" is the size of each element. The allocated memory is cleared to zeroes.

This function calls malloc() to allocate memory from a pre-defined arena. If the allocation is successful, calloc() returns a pointer to the area. If the allocation fails, 0 is returned.

SEE ALSO: malloc(), free(), ebrk(), _srqmem(), _srtmem().

ceil() Ceiling Function ceil()

Mathematical

SYNOPSIS:

```
double ceil(x)
double x;
```

DESCRIPTION: ceil() returns the smallest integer not less than "x".

SEE ALSO: floor().

chain() chainc() Load and Execute a New Module chain() chainc()

OS-9 System

SYNOPSIS:
```
chain(modname,parmsize,parmptr,type,lang,datasize,prior)
char *modname,*parmptr;
int parmsize,datasize;
short type,lang,prior;


chainc(modname,parmsize,parmptr,type,lang,datasize,prior,
       pathent)
char *modname,*parmptr;
int parmsize,datasize;
short type,lang,prior,pathent;
```

DESCRIPTION: chain() is used when it is necessary to execute an entirely new program without the overhead of creating a new process. It is functionally equivalent to an os9fork() followed by an exit(), but with less system overhead.

chain() effectively "resets" the calling process' program and data areas and begins execution of a new primary module. Open paths are not closed or otherwise effected.

The argument "modname" is a pointer to a null-terminated module name. The argument "parmptr" is a pointer to a null-terminated string to be passed to the new module. "parmsize" is the strlen() of "parmptr". The arguments "type" and "lang" specify the desired type and language of the module; these can be given as zero to indicate any type or language. The argument "datasize" gives extra memory to the new program. If no extra memory is required, this value can be zero. Finally, the argument "prior" is the new priority at which to run the program. Zero can be given if no priority change is desired.

If the chain is unsuccessful, chain() returns -1 and the appropriate error code is placed in the global variable "errno". If successful, chain() does not return; the new program begins execution.

chainc() is the same as chain() with a pathent argument. Pathent is the number of open paths for the new process to inherit. os9exec() is the preferred method by which to chain to C programs. See the discussion of os9exec() for more details.

Load and Execute a New Module

OS-9 System
(continued)

CAVEATS: Beware of chaining to a system object module. Usually, it only makes sense to chain to a program of type "object module".

SEE ALSO: The discussion of F$Chain and F$Fork in the "OS-9/68000 Operating System Technical Manual" and the discussion of **os9fork()**.

chdir() **Change Current Data Directory** **chdir()**

UNIX System

SYNOPSIS:
```
chdir(dirname)
char *dirname;
```

DESCRIPTION: This function changes the current data directory for the calling process. The argument "dirname" is a pointer to a string that gives a pathname for a directory.

chdir() returns 0 after a successful call. If "dirname" is not a directory pathname, a value of -1 is returned and the appropriate error code is placed in the global variable "errno".

SEE ALSO: The discussion of I$ChgDir in the "OS-9/68000 Operating System Technical Manual" and the OS-9 shell command "chd".

Change File Access Permissions

UNIX System

SYNOPSIS:

```
#include <modes.h>

chmod(name,perm)
char *name;
short perm;
```

DESCRIPTION: chmod() changes the access permission bits associated with a file. The argument "name" must be a pointer to a string containing a file name, and "perm" should contain the desired bit pattern.

chmod() returns 0 after a successful call. If the caller is not entitled to change the access permissions or the file can not be found, -1 is returned and the appropriate error code is placed in the global variable "errno".

CAVEATS: Only the super user or the owner of the file may change the access permissions.

SEE ALSO: The discussion of I$SetStt, function code SS_Attr in the "OS-9/68000 Operating System Technical Manual", the OS-9 command "attr", and the function _ss_attr().

chown() Change Owner of a File chown()

UNIX System

SYNOPSIS:

```
chown(name,newowner)
char *name;
short newowner;
```

DESCRIPTION: chown() changes the owner number of a file. The file name to be changed is pointed to by the argument "name", the new owner id is passed in "newowner".

chown() returns 0 after a successful call. If the caller is not entitled to change the owner id or the file can not be found, -1 is returned and the appropriate error code is placed in the global variable "errno".

CAVEATS: Only the super user may change the owner id.

SEE ALSO: The discussion of I$SetStt, function code SS_FD in the "OS-9/68000 Operating System Technical Manual" and the _ss_pfd() function.

chxdir() **Change Current Execution Directory** **chxdir()**

OS-9 System

SYNOPSIS:

```
chxdir(dirname)
char *dirname;
```

DESCRIPTION: This function changes the current execution directory for the calling process. The argument "dirname" is a pointer to a string that gives a pathname for a directory.

chdir() returns 0 after a successful call. If "dirname" is not a directory pathname, -1 is returned and the appropriate error code is placed in the global variable "errno".

SEE ALSO: The discussion of I$ChgDir in the "OS-9/68000 Operating System Technical Manual" and the OS-9 shell command "chx".

cleareof() Clear End of File Condition cleareof()

Standard I/O

SYNOPSIS:

```
#include <stdio.h>

cleareof(fp)
FILE *fp;
```

DESCRIPTION: This macro resets the end-of-file condition that causes feof() to return a non-zero value. This is useful to retry an input operation on a terminal after end-of-file is encountered. The getc() function will refuse to read characters until the end-of-file condition is cleared.

CAVEATS: Be sure to give cleareof() a file pointer and not a path number. cleareof() is implemented as a macro in <stdio.h>.

SEE ALSO: clearerr(), getc().

clearerr() Clear Error Condition **clearerr()**

Standard I/O

SYNOPSIS:

```
#include <stdio.h>

clearerr(fp)
FILE *fp;
```

DESCRIPTION: This macro resets the error condition that causes ferror() to return a non-zero value. This is useful to retry an input operation on a terminal after an error is encountered. The getc() function will refuse to read characters until the error condition is cleared.

CAVEATS: clearerr() resets the error condition on the file. This does not "fix" the file or prevent the error condition from occurring again. Be sure to give clearerr() a file pointer and not a path number. clearerr() is implemented as a macro in the <stdio.h> header file.

SEE ALSO: cleareof(), getc().

close() Close a Path close()

UNIX System

SYNOPSIS:

```
close(path)
int path;
```

DESCRIPTION: This function is used to close an open path. The path number is usually obtained by a previous call to open(), creat(), create(), or dup(). The standard paths 0, 1, and 2 (standard input, standard output, and standard error, respectively) are not normally closed by user programs.

If an error occurs during the close, this function returns -1 and the appropriate error code is placed in the global variable "errno".

CAVEATS: Be sure to use only a path number here, not a file pointer assigned by fopen().

SEE ALSO: open(), creat(), create(), dup().

closedir() **Closes the Named Directory Stream** **closedir()**

Miscellaneous

SYNOPSIS:

```
#include <dir.h>
closedir(dirp)
dir *dirp;
```

DESCRIPTION: Closedir() closes the named directory stream and frees the structure associated with dirp.

SEE ALSO: opendir(), readdir(), rewinddir(), seekdir() and telldir().

cos() Cosine cos()

Mathematical

SYNOPSIS:
```
#include <math.h>

double cos(x)
double x;
```

DESCRIPTION: cos() returns the cosine of "x" in radians.

crc() Calculate Module CRC crc()

OS-9 System

SYNOPSIS:

```
#include <module.h>

crc(ptr,count,accum)
char *ptr;
unsigned count;
int *accum;
```

DESCRIPTION: The crc() function generates the OS-9 CRC (cyclic redundancy check) value for use by compilers, assemblers, or other memory module generators.

The CRC is calculated starting at the byte pointed to by "ptr" for "count" bytes. "accum" is a pointer to the int CRC accumulator. It is not necessary to cover the entire module in one call as the CRC may be "accumulated" over several calls. The accumulator must be initialized to -1 before the first call.

When verifying the CRC of a module, initialize the accumulator to -1 and perform the CRC over the entire module, including the CRC bytes in the module. If the resulting CRC is equal to the CRC constant value the module is valid. A manifest constant CRCCON is defined in the <module.h> header file for this use.

To generate the CRC for a module, initialize the accumulator to -1, perform the CRC over the module, call crc() with a NULL value for "ptr", complement the CRC accumulator, and write the contents of the accumulator to the module.

The example on the next page illustrates a technique for generating a CRC for a module.

CAVEATS: The CRC value is three bytes long in a four byte field. To generate a valid module CRC, the caller must include the byte preceding the CRC in the calculation. The crc() function will assume a zero byte is to be CRCed if the argument "ptr" is passed as zero.

SEE ALSO: The F$CRC system call in the "OS-9/68000 Operating System Technical Manual".

crc() (continued)

EXAMPLE:

```
#include <stdio.h>
#include <module.h>

main()
{
    char *ptr;
    int count,accum = -1;

    .
    .
    .

    crc(ptr,count,&accum);    /* one or more calls to crc */

    .
    .
    .

    crc(ptr,count,&accum);
    /* above is the final call to crc().  At this point the
       entire module less the four crc bytes have been
       processed.
    */

    crc(NULL,0,&accum);
    /* above is a special case to run a null byte through
       the crc calculation.
    */

    accum = ~accum;
    /* complement the accumulator.  All previous calls have
       left the most significant byte as 0xff,
       complementing changes this byte to the required
       zero.
    */

    fwrite(&accum,1,sizeof accum,fp);
    /* finally write out the four crc bytes */

}
```

creat() **Create a File** creat()

UNIX System

SYNOPSIS:

```
#include <modes.h>

int creat(name,mode)
char *name;
short mode;
```

DESCRIPTION: creat() returns a path number to a new file with a name specified by a string pointed to by "name". The file is available for writing. The permissions are given by the argument "mode". The owner of the file is that of the task owner.

If, however, "name" is the name of an existing file, the file is truncated to zero length, and the ownership and permissions remain unchanged. The file will be open for write access.

If an error occurs, the value -1 is returned and the appropriate error code is placed in the global variable "errno".

NOTE: creat() does not return an error if the file exists. The access() function should be used to establish the existence of a file if it is important that a file should not be overwritten. The valid mode values are available in the <modes.h> header file.

CAVEATS: It is unnecessary to specify write permissions in "mode" to write to the file. Directories can not be created with this call. Instead, use the mknod() function.

SEE ALSO: The system calls I$Create and I$SetStt, function code _Size, the open() and create() functions.

create() Create a File create()

OS-9 System

SYNOPSIS:

```
#include <modes.h>

int create(name,mode,perm)
char *name;
short mode,perm;
```

DESCRIPTION: create() returns a path number to a new file with a name specified by the string pointed to by "name". The file is available for writing. The access permissions are given by the argument "mode". The file permission attributes are given in "perm". The owner of the file is the task owner.

If the file already exists or any other error occurs, -1 is returned and the appropriate error code is placed in the global variable "errno". The valid mode and permission values are available in the <modes.h> header file.

CAVEATS: This function is similar to the creat() call except it allows the caller to give the exact file attributes desired and does not truncate the file if it is already present. Directories can not be created with this call. Instead, use the mknod() function.

SEE ALSO: The system call I$Create and the open() and creat() functions.

detach() **Detach a Device** **detach()**

OS-9 System

SYNOPSIS:
```
detach(ptr)
char *ptr;
```

DESCRIPTION: detach() removes a device from the system device table if it is not in use by any other process. If this is the last use of the device, the device driver's termination routine is called. Any permanent storage assigned to the device is deallocated.

The argument "ptr" must be a pointer returned by the attach() call.

If an error occurs, the function returns -1 and the appropriate error code is placed in the global variable "errno".

SEE ALSO: The system call I$Detach and the attach() function.

dup() Duplicate a Path dup()

UNIX System

SYNOPSIS:
```
int dup(path)
int path;
```

DESCRIPTION: Given an existing path, dup() returns a synonymous path number for the same file or device. The lowest available path number is used.

dup() merely increments the "link count" of a path descriptor d returns a different synonymous path number. The path descriptor is not cloned.

NOTE: It is usually not a good idea for more than one process to be doing I/O on the same path concurrently. On RBF files, unpredictable results may occur.

SEE ALSO: The system call I$Dup in the "OS-9/68000 Operating System Technical Manual".

ebrk() Obtain External Memory ebrk()

Memory Management

SYNOPSIS:
```
char *ebrk(size)
unsigned size;
```

DESCRIPTION: This function returns a specified amount of memory ("size"). The memory is obtained from the system via the F$SrqMem system request. It is intended for general purpose memory allocation.

The blocks of memory returned by this call may not be contiguous, thereby providing the ability to obtain a block of memory of a given size from anywhere in the 68000 address space.

To reduce the overhead involved in requesting small quantities of memory, ebrk() will request memory from the system in a minimum size determined by the global variable "_memmins" which is initially set to 4096, and satisfy the user requests from this memory space. ebrk() will grant memory requests from this memory space provided the requests are no larger than the amount of space.

If the request is larger than the available space, ebrk() will waste the rest of the space and try to get enough memory from the system to satisfy the request.

This method works very well for programs that need to get large amounts of not necessarily contiguous memory in little bits and can not afford the overhead of malloc().

Changing the "_memmins" variable will cause ebrk() to use that value as the F$SrqMem memory request size.

If the memory request is granted, a pointer (even-byte aligned) to the block is returned. If the request is not granted, -1 is returned and the appropriate error code is placed in the global variable "errno".

CAVEATS: The memory obtained from ebrk() is not given back until the process terminates.

SEE ALSO: The system call F$SrqMem system call in the "OS-9/68000 Operating System Technical Manual" and the functions sbrk(), ibrk(), malloc().

exit() Terminate Task exit()

UNIX System

SYNOPSIS:
```
exit(status)
short status;
```

DESCRIPTION: exit() is the normal means of terminating a task. exit() does any "clean up" operations required before terminating, such as flushing out any file buffers. The "status" word passed is available to the parent task if it is executing a wait(). Hopefully, exit() returns no value...

SEE ALSO: The system call F$Exit system call in the "OS-9/68000 Operating System Technical Manual" and the functions _exit().

exp() **Exponential Function** **exp()**

Mathematical

SYNOPSIS:
```
double exp(x)
double x;
```

DESCRIPTION: exp() returns the value e (2.71828...) raised to the power "x".

fabs() Floating Absolute Value **fabs()**

Mathematical

SYNOPSIS:
```
double fabs(x)
double x;
```

DESCRIPTION: fabs() returns the absolute value of the argument "x".

fclose() **Close a File** **fclose()**

Standard I/O

SYNOPSIS:

```
#include <stdio.h>

int fclose(fp)
FILE *fp;
```

DESCRIPTION: fclose() flushes the buffer associated with file pointer "fp". It closes the file and frees the buffer for use by another call to fopen().

It is not considered an error to fclose() a file pointer that is not open, but it is required that "fp" be obtained from fopen(), or be one of the predefined macros "stdin", "stdout", or "stderr".

The function returns EOF if an error occurs during the close.

SEE ALSO: fopen(), fflush(), getc(), and putc().

fdopen() Attach a Path to a File Pointer fdopen()

Standard I/O

SYNOPSIS:
```
#include <stdio.h>

FILE *fdopen(path,action)
int path;
char *action;
```

DESCRIPTION: fdopen() associates a currently open path with a file pointer. The "action" given must be compatible with the access mode of the path given. The discussion of fopen() details the valid action values.

This function is useful when some special processing is required when opening the file that fopen() does not provide.

SEE ALSO: fopen(), freopen().

feof() Check Buffered File for End of File feof()

Standard I/O

SYNOPSIS:

```
#include <stdio.h>

int feof(fp)
FILE *fp;
```

DESCRIPTION: feof() is a macro used to test a file to see if it is at end-of-file. The argument "fp" is a pointer to a FILE structure (not a path number).

A non-zero value is returned if the file is at end-of-file, otherwise a zero is returned.

The macro cleareof() can be used to clear the end-of-file condition.

SEE ALSO: fopen(), cleareof().

ferror() Check Buffered File for Error Condition ferror()

Standard I/O

SYNOPSIS:

```
#include <stdio.h>

int ferror(fp)
FILE *fp;
```

DESCRIPTION: ferror() is a macro used to test a file to see if an error occurred. The argument "fp" is a pointer to a FILE structure ([illegible]t a path number).

A non-zero value is returned if an error occurred from the last I/O operation, otherwise a zero is returned.

The macro clearerr() can be used to clear the error condition.

CAVEATS: There is really no way to tell what error occurred or when.

SEE ALSO: fopen(), clearerr().

fflush() **Flush a File's Buffer** **fflush()**

Standard I/O

SYNOPSIS: #include <stdio.h>

```
int fflush(fp)
FILE *fp;
```

DESCRIPTION: fflush() causes a buffer associated with the file pointer "fp" to be cleared by writing out to the file. The file will be written to only if it was opened for write or update.

It is not normally necessary to call **fflush()**, but it can be useful if the terminal is open on a path other than one of the standard paths and the buffered output needs to be flushed before input.

If a getc() (or equivalent) is performed on "stdin", the "stdout" buffer will be flushed automatically.

The function **fclose()** calls **fflush()** to clean out the buffers before the file is closed.

SEE ALSO: fopen(), fflush(), getc(), putc().

fgetc() Get a Character from a File **fgetc()**

Standard I/O

SYNOPSIS:
```
#include <stdio.h>

int fgetc(fp)
FILE *fp;
```

DESCRIPTION: fgetc() returns a character from a file pointed to by "fp". EOF is returned on end-of-file or error.

In this implementation, fgetc() is a macro which calls getc() which is a genuine function. UNIX normally defines getc() as a macro and fgetc() as a genuine function. The usage and result is the same on both systems.

SEE ALSO: getc().

fileno() Determine Path Number from File fileno()

Standard I/O

SYNOPSIS:

```
#include <stdio.h>

int fileno(fp)
FILE *fp;
```

DESCRIPTION: fileno() returns the path number associated with file pointer "fp". The path number is not valid unless (fp->_flag & _INIT) is non-zero.

findnstr() **Search String for Pattern** **findnstr()**

String Handling

SYNOPSIS:
```
findnstr(pos,string,pattern,size)
int pos,size;
char *string,*pattern;
```

DESCRIPTION: This function will search the string pointed to by "string" for the first instance of the pattern pointed to by "pattern". It will start at position "pos" (where the first position is 1, not zero). The returned value is the position of the first matched character of the pattern in the string, or zero if a match is not found. findnstr() will stop searching only at position "pos" + "len" so it may continue past null bytes.

CAVEATS: The current implementation does not use the most efficient algorithm for pattern matching. Use on very long strings is likely to be somewhat slower than it should be.

SEE ALSO: findstr().

findstr() Search String for Pattern findstr()

String Handling

SYNOPSIS:

```
findstr(pos,string,pattern)
int pos;
char *string,*pattern;
```

DESCRIPTION: This function will search the string pointed to by "string" for the first instance of the pattern pointed to by "pattern". It will start at position "pos" (where the first position is 1, not zero). The returned value is the position of the first matched character of the pattern in the string, or zero if a match is not found. findstr() will stop searching when a null byte is found in "string".

CAVEATS: The current implementation does not use the most efficient algorithm for pattern matching. Use on very long strings is likely to be somewhat slower than it should be.

SEE ALSO: findnstr().

floor() **Floor Function** **floor()**

Mathematical

SYNOPSIS:

```
#include <math.h>

double floor(x)
double x;
```

DESCRIPTION: floor() returns the largest integer (as a double) that is not greater than "x".

fopen() Open a File fopen()

Standard I/O

SYNOPSIS:
```
#include <stdio.h>

FILE *fopen(name,action)
char *name,*action;
```

DESCRIPTION: fopen() returns a pointer to a structure describing a buffered file if the string pointed to by "name" is a file that can be validly opened with the action in the string pointed to by "action".

The valid fopen() actions are:

ACTION	DESCRIPTION
"r"	open for reading
"w"	open for writing
"a"	append (write) at the end of the file, or create file for writing if "name" does not exist
"r+"	open for update
"w+"	create for update
"a+"	create or open for update at end of file
"d"	directory read

NOTE: Any action may have an "x" after the initial letter. This indicates to fopen() that it should look in the current execution directory if a full pathlist is not given. The "x" also implies that the file should have the execute permission bit set.

EXAMPLES:
```
fp = fopen("fred","wx");     /* creates file for writing in
                                the execution directory */
fp = fopen("moe","r");       /* opens existing file for
                                reading*/

fp = fopen("stooge/curly","w+"); /* creates file for both
                                    reading and writing */
```

fopen() (continued)

CAVEATS: Make sure the argument passed as "action" is a pointer to a string and not a character; fopen("fun","r") is correct, fopen("fun",'r') is not.

Opening for write will perform a creat(). If a file with the same name exists when the file is opened for write, it will be truncated to zero length. Only if the file does not exist will it be created.

Append means open for write and position the file pointer to the end-of-file. This will cause the next byte written to be added to the end of the file. If a read is performed, it will return an end-of-file error. Note that the type of a file structure is pre-defined in <stdio.h> as FILE, so a user program may declare or define a file pointer by:

```
FILE *f;
```

Three file pointers are available and can be considered open the moment the program runs:

stdin	the standard input	I/O is to path 0
stdout	the standard output	I/O is to path 1
stderr	the standard error output	I/O is to path 2

These macros are defined in the header file <stdio.h>.

SEE ALSO: **putc(), getc(), creat(), open().**

fprintf() Formatted Output fprintf()

Standard I/O

SYNOPSIS:

```
int fprintf(fp, control [,arg0[,arg1...]])
FILE *fp;
char *control;
```

DESCRIPTION: printf(), fprintf(), and sprintf() are standard C library functions that perform formatted output. Each of these functions converts, formats, and prints the "args" (if any) as indicated by the "control" string.

fprintf() places its output on the file pointed to by "fp". See the discussion of printf() for details on the control string.

fprintf() returns the number of characters output.

CAVEATS: ferror() can be used to check for an output error after calling fprintf().

SEE ALSO: printf(), sprintf().

fputs() Output a String to a File **fputs()**

Standard I/O

SYNOPSIS:

```
fputs(ptr,fp)
char *ptr;
FILE *fp;
```

DESCRIPTION: fputs() copies the null terminated string pointed to by "ptr" onto the file "fp". The string terminating zero byte is not copied. -1 is returned if an output error occurred.

SEE ALSO: puts(), gets(), fgets().

fread() Read Data from a File fread()

Standard I/O

SYNOPSIS:

```
#include <stdio.h>

int fread(ptr, size, nobj, fp)
char *ptr;
int size, nobj;
FILE *fp;
```

DESCRIPTION: fread() reads from the file pointed to by "fp". The argument "nobj" is the number of items to be read. Each item is of size "size" bytes. The bytes read are copied into the memory pointed to by "ptr. fread() returns the number of items read, or zero if an error occurs.

EXAMPLE:

```
func()
{
   int value;
   FILE *file;

   .
   .
   .

   fread(&value, sizeof value, 1, file);

}
```

CAVEATS: It is the responsibility of the caller to ensure enough space is available at "ptr".

SEE ALSO: fwrite().

free() **Return Memory to Arena** **free()**

Memory Management

SYNOPSIS:
```
free(ptr)
char *ptr;
```

DESCRIPTION: free() returns a block of memory granted by malloc() or calloc(). The memory is returned to a pool of memory for later re-use by malloc() or free(). The memory is not returned to the system.

CAVEATS: It is dangerous to use free() with something other than a pointer previously returned by malloc() or calloc(). To do so will hopelessly corrupt the memory arena, rendering malloc() useless or possibly causing unpredictable program behavior.

SEE ALSO: malloc(), calloc().

freemem() Determine Size of Unused Stack Area freemem()

Miscellaneous

SYNOPSIS: int freemem()

DESCRIPTION: freemem() returns the number of bytes allocated for the stack that have not been used.

If compiler stack checking is enabled, the stack is checked for possible overflow before a function is entered. The lowest address the stack pointer has reached is retained so freemem() can report the number of bytes between the stack limit and the lowest stack value as the "unused" stack memory.

CAVEATS: The program must be compiled with stack checking code in effect for freemem() to return a correct result.

SEE ALSO: stacksiz().

freopen() **Re-Open a File** **freopen()**

Standard I/O

SYNOPSIS:

```
#include <stdio.h>

FILE *freopen(name,action,fp)
char *name,*action;
FILE *fp;
```

DESCRIPTION: **freopen()** is usually used to associate the "stdin", "stdout", or "stderr" file pointers with a file instead of a terminal device.

freopen() substitutes the file name passed as "name" instead of the currently open file (if any). The original file is closed with **fclose()**. The original file will be closed even if the open of the new file does not succeed.

See the discussion for **fopen()** for details on the "action" values.

freopen() returns the "fp" passed or zero if the open failed.

SEE ALSO: **fopen()**, **fdopen()**.

fscanf() Input String Conversion fscanf()

Standard I/O

SYNOPSIS:

```
#include <stdio.h>

int fscanf(fp, control [,arg ...])
FILE *fp;
char *control;
```

DESCRIPTION: **fscanf()** performs input conversions from data read from the file pointer "fp". The format of the "control" string is described in **scanf()**. **fscanf()** returns a count of the number of fields successfully matched and assigned.

See the discussion of **scanf()** for more details.

SEE ALSO: **scanf()**, **sscanf()**.

fseek() Reposition File Pointer **fseek()**

Standard I/O

SYNOPSIS:

```
#include <stdio.h>

int fseek(fp, offset, place)
FILE *fp;
long offset;          /* long (same as int) */
int place;
```

DESCRIPTION: fseek() repositions the file pointer for the buffered file pointed to by "fp" to a desired character position for the next getc() or putc(). The new position is at "offset" bytes from:

the beginning of the file if "place" is 0,
the current position in the file if "place" is 1, or
the end of the file if "place" is 2.

fseek() allows for the special problems of buffering. fseek() returns 0 if a seek is reasonable or -1 if the destination (or place) is otherwise incorrect.

CAVEATS: Using lseek() of a buffered file is certain to cause unpredictable results. fseek() is required when changing from reading to writing or from writing to reading on files open for update.

SEE ALSO: getc(), putc(), lseek().

ftell() Report File Pointer Position ftell()

Standard I/O

SYNOPSIS: #include <stdio.h>

```
long ftell(fp)
FILE *fp;
```

DESCRIPTION: ftell() is the only proper way to determine the next byte to be read or written when using buffered I/O. ftell() returns the current position of the next byte to be read by getc() or written by putc(). The current position is measured in bytes from the beginning of the file.

"fp" must be a file pointer.

-1 is returned if "fp" does not point to an open file.

CAVEATS: Do not use lseek() to determine file position as it does not allow for any characters in the buffer.

SEE ALSO: getc(), putc(), lseek().

fwrite() Write Data to a File fwrite()

Standard I/O

SYNOPSIS:

```
#include <stdio.h>

int fwrite(ptr, size, nobj, fp)
char *ptr;
int size, nobj;
FILE *fp;
```

DESCRIPTION: fwrite() writes to the file pointed to by "fp". The argument "nobj" is the number of items to be written. Each item is of size "size" bytes. The bytes written are copied from the memory pointed to by "ptr". fwrite() returns the number of items written, or zero if an error occurs.

EXAMPLE:

```
func()
{
   int value;
   FILE *file;

   .
   .
   .

   fwrite(&value, sizeofvalue, 1, file);

}
```

SEE ALSO: fread().

getc() Get Next Character from File getc()

Standard I/O

SYNOPSIS:

```
#include <stdio.h>

int getc(fp)
FILE *fp;
```

DESCRIPTION: getc() returns the next character from the file pointed to by "fp". The value returned is an int not a char. This is so the caller can tell the difference between a 0xff byte read from the file and the error condition -1, which is returned when an error or end-of-file occurs. The appropriate error code is left in the global variable "errno" when an error occurs.

Under OS-9, there is a choice of low-level service requests to use when reading from a file:

read() will get characters up to a specified bytecount in "raw" mode (i.e., no editing will take place on the input stream and the characters will appear to the program exactly as in the file or on the device).

readln() on the other hand, will honor the various mapping of characters associated with a Serial Character device such as a terminal and in any case will return to the caller as soon as a carriage return is seen.

In the vast majority of cases, it is preferable to use readln() for accessing Serial Character devices and read() for any other file input. getc() uses this strategy. As all file input using the Standard Library function is routed through getc(), so do all other library input functions.

The choice is made when the first call to getc() is made after the file is opened. The system is consulted for the status of the file and a flag bit is set in the file structure accordingly.

The choice may be forced by the programmer by setting the relevant bit before a call to getc(). The flag bits are defined in the <stdio.h> header file as _RBF and _SCF.

getc() (continued)

The following method is used to set the flag bits. Assuming the file pointer (as returned by fopen()) is fp, the following will force the use of readln() and read() on input, respectively:

```
        fp->_flag |= _SCF;

        fp->_flag |= _RBF;
```

This trick may be played on the standard stream, "stdin", before any input is requested from the stream.

When it is desired to input characters one byte at a time without buffering, use this technique:

```
        fp->_flag |= _UNBUF;
```

This will cause data to be read from or written to the file one byte at a time.

If a file is open for update (fopen() action "r+" or "w+") the program is required to do an **fseek()** before changing from getc() to putc() or putc() to getc(), even if no effective file position change occurs. This is done to cause the buffer to be flushed (or filled) so input or output can proceed in the opposite direction.

The "stdout" buffer is flushed automatically by getc(stdin) before the actual read is performed. This is so that prompts to a erminal are sure to be output before the read is issued.

CAVEATS: It is not a good idea to call read() or write() on a path that is buffered since the buffered data and the low-level I/O may not occur in the correct order.

NOTE: Due to the buffering, RBF record-locking is ineffective. If an application requires the record-locking facilities, low-level I/O calls and program buffering is required.

SEE ALSO: fopen(), putc().

getchar() **Get Next Character from stdin** **getchar()**

Standard I/O

SYNOPSIS: #include <stdio.h>

int getchar()

DESCRIPTION: getchar() is a macro that is equivalent to getc(stdin).

SEE ALSO: getc().

getenv() **Value for Enviroment Name** getenv()

UNIX System

SYNOPSIS: `char *getenv(name)`
`char *name;`

DESCRIPTION: Getenv() searches the enviroment list for a string in the form of "name = value". It returns a pointer to the string value if such a string is present, otherwise **getenv()** returns the value 0 (NULL).

An array of strings called the enviroment is made available by os9exec() when a process is created. The enviroment list is usually maintained by the Shell. The strings can be altered by the Shell setenv and unsetenv commands. The names of the enviroment variables and their contents are defined by the application using them.

NOTE: A process will inherit the enviroment only if the process was created via the os9exec() function. The os9fork() and chain() functions themselves do not pass the correct information for the argument and enviroment lists.

SEE ALSO: os9exec(), os9fork() and chain()

getime() Get System Time getime()

UNIX System

SYNOPSIS:

```
#include <time.h>

getime(timebuf)
struct sgtbuf *timebuf;
```

DESCRIPTION: getime() returns the system time into the time buffer pointed to by "timebuf". The time units are defined in the <time.h> header file.

SEE ALSO: setime().

getpid() **Determine Process ID Number** **getpid()**

UNIX System

SYNOPSIS: int getpid()

DESCRIPTION: getpid() returns the system process ID number for the calling process. This number is useful for such things as creating unique file names.

SEE ALSO: The discussion of F$ID in the "OS-9/68000 Operating System Technical Manual".

gets() **Get a String from a File** **gets()**

Standard I/O

SYNOPSIS: `#include <stdio.h>`

```
char *gets(ptr)
char *ptr;
```

DESCRIPTION: gets() reads characters from the "stdin" file and places them into the buffer pointed to by "ptr". The function fills the buffer until a carriage return ('\n') is read. The '\n' is replaced by a null byte. gets() returns its argument.

CAVEATS: Since no maximum bytecount is available for gets(), it is the responsibility of the caller to reserve enough bytes at "ptr" for the string.

SEE ALSO: fgets().

getstat() **Get File Status** **getstat()**

OS-9 System

SYNOPSIS:

```
#include <sgstat.h>

getstat(code, path, buffer)               /* code == 0 */
int code,path;
char *buffer;

getstat(code,path)                        /* code == 1 or 6 */
int code,path;

getstat(code,path,size)                   /* code == 2 */
int code,path;
long *size;

getstat(code,path,pos)                    /* code == 5 */
int code,path;
long *pos;
```

DESCRIPTION: This function is historical from the 6809 C Compiler and is used to access a few I$GetStt system functions. The argument "path" must be the path number of an open file. The argument "code" is defined as follows:

code = 0	"buffer" is the address of the 128 byte buffer into which the path option bytes are copied. The <sgstat.h> header file contains a struct defined for use by the program.
code = 1	This applies only to SCF devices and is used to test for data ready. The return value is the number of bytes available or -1 if an error occurred.
code = 2	"size" is the address of the long int into which the current file size is placed. The function returns -1 on error and zero on success.
code = 5	"pos" is the address of the long int into which the current file position is placed. The function returns -1 on error and zero on success.
code = 6	The function returns -1 on either EOF or error and 0 if not at EOF.

getstat() (continued)

When getstat() returns with the value -1, the appropriate error code is placed in the global variable "errno".

CAVEATS: This function is supported for the convenience of programs ported from the 6809. See the special getstat functions (all of which begin with _gs) for the same information supplied in a more palatable format.

SEE ALSO: setstat(), all the "_gs" functions (see index).

getuid() **Determine User ID Number** **getuid()**

UNIX System

SYNOPSIS: `int getuid()`

DESCRIPTION: getuid() returns the group/user id of the current process. The upper word (2 bytes) of the value is the group number, the lower word is the user number.

SEE ALSO: The discussion of F$ID in the "OS-9/68000 Operating System Technical Manual".

getw() Read a Word from a File getw()

Standard I/O

SYNOPSIS:

```
#include <stdio.h>

int getw(fp)
FILE *fp;
```

DESCRIPTION: getw() returns an integer from the file pointed to by "[illegible]". Since it returns -1 on error, the macros feof() and ferror() should be used to check the success of getw().

CAVEATS: getw() is a very machine-dependent function since the size of a "word" varies from machine to machine. getw() assumes no particular alignment in the file.

hypot() **Euclidean Distance Function** **hypot()**

Mathematical

SYNOPSIS:

```
#include <math.h>

double hypot(x, y)
double x, y;
```

DESCRIPTION: hypot() returns the Euclidean distance function:

```
sqrt( x * x + y * y),
```

It takes precautions against unwarranted overflows.

ibrk() Request Internal Memory ibrk()

Memory Management

SYNOPSIS: char *ibrk(size)
unsigned size;

DESCRIPTION: ibrk() returns a pointer to a block of memory of size "size". The returned pointer is aligned to a word boundary. The memory from which ibrk() grants requests is the area between the end of the data allocation and the stack:

```
Higher Address:       .------------------.
                      |                  |    Grows toward
                      |    Stack Area    |  | Ibrk Area
Stack Pointer     ->  |                  |  v

Stack Limit       ->  |                  |
changes as ibrk       |                  |
area grows            |    Ibrk Area     |  ^
                      |                  |  | Grows toward
                      >------------------<    Stack
                      |                  |
                      |     Program      |
                      |    Data Area     |
                      |                  |
Lowest Address:       '------------------'
```

If the requested size would cause the ibrk area to cross the stack pointer, the request fails. The **freemem()** function can be used to determine the amount of stack remaining which is also the remaining ibrk area.

ibrk() is useful to obtain memory from a fixed amount of memory, unlike ebrk() whose available memory is that of the entire system. The C I/O system will request the first 2K of I/O buffers from this area, the remainder from ebrk().

CAVEATS: Be very careful so as not to "crowd out" the stack with ibrk() calls. When stack checking is in effect, the program will abort with a "***Stack Overflow***" message, if insufficient stack area exists to call a function.

SEE ALSO: sbrk(), ebrk(), freemem(), stacksize().

index() Search for Character in String **index()**

String Handling

SYNOPSIS:
```
char *index(ptr,ch)
char *ptr;
char ch;
```

DESCRIPTION: index() returns a pointer to the first occurrence of the character "ch" in the string pointed to by "ptr". If the character is not found, the function returns a NULL (0).

CAVEATS: Note that "ch" is a character, not a pointer to a character:

```
func()
{
    char *ptr,*string;

    .
    .
    .

    if((ptr = index(string,'.')) {
          process(ptr);
    } else {
          printf("No '.' found!\n");
    }
}
```

The above example looks for a '.' (period) in the string string" and sets "ptr" to point to it.

SEE ALSO: rindex().

intercept() Set up Process Signal Handler intercept()

OS-9 System

SYNOPSIS:

```
int intercept(icpthand)
int (*icpthand)();      /* this is how a function declares*/
                        /* that an argument is a pointer */
                        /* to a function returning an int */
```

DESCRIPTION: intercept() instructs OS-9 to pass control to the function whose address is pointed to by "icpthand" when a signal is received by the process.

If the signal handler function declares an int argument, the function will have access to the number of the signal received. On return from the signal handler function, the process resumes at the point in the program where it was interrupted by the signal.

Since signals do not "queue up" for a process, it is desirable to keep signal handler functions as short as possible so they can process signals as fast as possible. Ideally, the signal handler should set a global flag for the mainline to periodically examine and let the mainline do the real work required.

As an example, suppose that a program wants to clean up work files and exit when a keyboard quit signal (signal 3 which, normally, can be caused by typing ^E on the terminal). See the next page for an example program section.

CAVEATS: A program will not receive an abort or quit signal from the keyboard (usually ^C and ^E) unless the program does output to the terminal. This is because OS-9 sends the abort/quit signals to the last process to do I/O to the terminal. If you run the program from the shell and type ^E before the program does I/O to the terminal, the shell will receive the signal and kill the running program. If a program requires control of the terminal immediately, do some I/O to one of the standard paths such as printing a program banner or getting the terminal options with _gs_opt().

SEE ALSO: The F$Icpt system request in the "OS-9/68000 Operating System Technical Manual".

intercept() (continued)

EXAMPLE: Example program segment that uses intercept():

```
#include <stdio.h>

#define until(expr)      while(!(expr))

char *tempname = "tempfile";
FILE *tempfp;
int quittime = 0;

/* the signal handler */
gotsignl(signum)
int signum;
{
      case 3:                  /* the quit signal */
         quittime = 1;
         break;
      default:                 /* ignore all others */
         break;
}

main()
{
     if((tempfp = fopen(tempname,"w") == NULL)
         exit(_errmsg(errno,"can't open file - %s\n",tempname);

    do {
        do_work();
    } until(quittime);

    fclose(tempfp);
    unlink(tempname);
    exit(_errmsg(1,"quittin' time!!!\n"));
}
```

isalnum() See if Argument is Alphanumeric **isalnum()**

Character Classification

SYNOPSIS: **#include <ctype.h>**

```
int isalnum(c)
char c;
```

DESCRIPTION: The macro **isalnum()** returns a non-zero value if its argument is a digit or an alphabetic character. Otherwise it returns zero.

CAVEATS: The domain of this macro is defined only for "c" = -1 (EOF) and the values of "c" that cause **isascii()** to return a non-zero value.

SEE ALSO: **isascii()**.

isalphа() See if Argument is Alphabetic **isalpha()**

Character Classification

SYNOPSIS: `#include <ctype.h>`

```
int isalpha(c)
char c;
```

DESCRIPTION: The macro isalpha() returns a non-zero value if its argument is an alphabetic character. Otherwise it returns zero.

CAVEATS: The domain of this macro is defined only for "c" = -1 (EOF) and the values of "c" that cause isascii() to return a non-zero alue.

SEE ALSO: isascii().

isascii() **See if Argument is ASCII** **isascii()**

Character Classification

SYNOPSIS: **#include <ctype.h>**

```
int isascii(c)
char c;
```

DESCRIPTION: The macro **isascii()** returns a non-zero value if its argument is an ASCII character; i.e., the value is less than 128. Otherwise it returns zero.

SEE ALSO: **isalnum()**, **isalpha()**, **iscntrl()**, **isdigit()**, **islower()**, **isprint()**, **ispunct()**, **isspace()**, **isupper()**, **isxdigit()**.

isentrl() **See if Argument is a Control Code** **isentrl()**

Character Classification

SYNOPSIS: **#include <ctype.h>**

```
int isentrl(c)
char c;
```

DESCRIPTION: The macro **isentrl()** returns a non-zero value if its argument is a control code (values 0 through 31 and 127). Otherwise it returns zero.

CAVEATS: The domain of this macro is defined only for "c" = -1 (EOF) and the values of "c" that cause **isascii()** to return a non-zero value.

SEE ALSO: isascii().

isdigit() See if Argument is a Digit isdigit()

Character Classification

SYNOPSIS:
```
#include <ctype.h>

int isdigit(c)
char c;
```

DESCRIPTION: The macro **isdigit()** returns a non-zero value if its argument is a digit. Otherwise it returns zero.

CAVEATS: The domain of this macro is defined only for "c" = -1 (EOF) and the values of "c" that cause **isascii()** to return a non-zero value.

SEE ALSO: **isascii()**.

islower() **See if Argument is Lower Case** **islower()**

Character Classification

SYNOPSIS: **#include <ctype.h>**

```
int islower(c)
char c;
```

DESCRIPTION: The macro **islower()** returns a non-zero value if its argument is a lower case letter. Otherwise it returns zero.

CAVEATS: The domain of this macro is defined only for "c" = -1 (EOF) and the values of "c" that cause **isascii()** to return a non-zero value.

SEE ALSO: **isascii()**.

isprint() See if Argument is a Printable Character isprint()

Character Classification

SYNOPSIS: `#include <ctype.h>`

```
int isprint(c)
char c;
```

DESCRIPTION: The macro **isprint()** returns a non-zero value if its argument is a printable character (values 32 through 126). Otherwise it returns zero.

CAVEATS: The domain of this macro is defined only for "c" = -1 (EOF) and the values of "c" that cause **isascii()** to return a non-zero value.

SEE ALSO: **isascii()**.

ispunct() **See if Argument is a Punctuation Character** **ispunct()**

Character Classification

SYNOPSIS: **#include <ctype.h>**

```
int ispunct(c)
char c;
```

DESCRIPTION: The macro **ispunct()** returns a non-zero value if its argument is a punctuation character (neither control nor alphanumeric). Otherwise it returns zero.

CAVEATS: The domain of this macro is defined only for "c" = -1 (EOF) and the values of "c" that cause **isascii()** to return a non-zero value.

SEE ALSO: **isascii()**.

isspace() **See if Argument is White Space** **isspace()**

Character Classification

SYNOPSIS:

```
#include <ctype.h>

int isspace(c)
char c;
```

DESCRIPTION: The macro **isspace()** returns a non-zero value if its argument is one of the white space characters (space, tab, linefeed, return, or formfeed). Otherwise it returns zero.

CAVEATS: The domain of this macro is defined only for "c" = -1 (EOF) and the values of "c" that cause **isascii()** to return a non-zero value.

isupper() See if Argument is Upper Case isupper()

Character Classification

SYNOPSIS:

```
#include <ctype.h>

int isupper(c)
char c;
```

DESCRIPTION: The macro isupper() returns a non-zero value if its argument is an upper case letter. Otherwise it returns zero.

CAVEATS: The domain of this macro is defined only for "c" = -1 (EOF) and the values of "c" that cause **isascii()** to return a non-zero value.

SEE ALSO: isascii().

isxdigit() **See if Argument is a Hex Character** **isxdigit()**

Character Classification

SYNOPSIS: **#include <ctype.h>**

```
int isxdigit(c)
char c;
```

DESCRIPTION: The macro **isxdigit()** returns a non-zero value if its argument is a hexadecimal digit (0-9, A-F, or a-f). Otherwise it returns zero.

CAVEATS: The domain of this macro is defined only for "c" = -1 (EOF) and the values of "c" that cause **isascii()** to return a non-zero value.

SEE ALSO: **isascii().**

kill() Send a Signal to a Process kill()

UNIX System

SYNOPSIS:

```
#include <signal.h>

int kill(pid,sigcode)
int pid;
short sigcode;
```

DESCRIPTION: kill() is used to send a signal to a process. Both the sending and receiving process must have the same user number unless the sending process' user number is that of the super user (0).

The value in "sigcode" is sent as a signal to the process whose ID number is "pid". Any value can be given in "sigcode". The conventional code numbers are defined in the <signal.h> header file.

kill() returns -1 if an error occurs and the appropriate error code is placed in the global variable "errno".

CAVEATS: The super user can send a signal to all processes running on the system if the "pid" is zero.

SEE ALSO: The F$Send service request in the "OS-9/68000 Operating System Technical Manual" and the shell's "kill" command.

log() Natural Logarithm log()

Mathematical

SYNOPSIS: #include <math.h>

```
double log(x)
double x;
```

DESCRIPTION: log() returns the natural logarithm of "x". The value of "x" must be positive.

log10() Natural Logarithm Base Ten log10()

Mathematical

SYNOPSIS:

```
#include <math.h>

double log10(x)
double x;
```

DESCRIPTION: log() returns the natural logarithm base ten of "x". The value of "x" must be positive.

Miscellaneous

SYNOPSIS:

```
#include <setjmp.h>

int longjmp(env,val)
jmp_buf env;
int val;
```

DESCRIPTION: setjmp() and longjmp() allow the return of program control directly to a higher level function. They are most useful when dealing with errors and signals encountered in a low-level routine.

The "goto" statement in C is limited in scope to the function in which it appears (i.e., the destination of the "goto" must be in the same function). Control can only be transferred elsewhere by means of the function call, which of course, returns to the caller. In certain abnormal situations a programmer would prefer to be able to start some section of the code again. But this would mean returning up a ladder of function calls with error indications all the way.

setjmp() is used to "mark" a point in the program where a subsequent longjmp() can reach. setjmp() places in the buffer passed in "env" (as defined in the <setjmp.h> header file) enough information for longjmp() to restore the environment to that existing at the associated call to setjmp().

longjmp() is called with the environment buffer as an argument and a value "val" can be used by the caller of longjmp() as, perhaps, an error status indicator.

To set the setjmp() facility up, a function will call setjmp() to initialize the buffer, and if the value returned by setjmp() is zero, the program will know that the call was the "first time through". If, however, the returned value was non-zero, it must have been a longjmp() returning from some deeper level of the program.

After longjmp() is completed, program execution continues as if the corresponding setjmp() call had just returned the value "val". It is imperative that the function calling setjmp() does not return before the call to longjmp().

longjmp() can not cause setjmp() to return the value zero as that value is returned by the call to setjmp() itself. If longjmp() is invoked with a "val" of zero, setjmp() will return 1. All automatic variables have values as of the time longjmp() was called.

longjmp() (continued)

CAVEATS: If longjmp() is called before "env" is initialized by **setjmp()**; or when the function which made the last call to **setjmp()** has since returned, absolute chaos is guaranteed.

SEE ALSO: setjmp().

lseek() Position File Pointer lseek()

UNIX System

SYNOPSIS:

```
long lseek(path,position,place)
int path;
long position;
int place;
```

DESCRIPTION: **lseek()** repositions the file pointer for the file open on "path", to the byte offset given in "position". The argument "place" controls from where the offset is based: if 0, from the beginning of the file, if 1, from the current position, or if 2, from the end of the file.

Seeking to a location beyond end-of-file for a file open for writing and then writing to it creates a "hole" in the file. The hole will contain data with no particular value, usually garbage remaining on the disk.

The value returned is the resulting position in the file. If there is an error, -1 is returned and the appropriate error code is placed in the global variable "errno".

CAVEATS: On OS-9 the file pointer is a full unsigned 32-bits. UNIX file addresses are limited to 31 bits, and an error is returned if the resulting file pointer would be negative.

Do not use **lseek()** on a buffered file since the buffering routines keep track of the file pointer via **fseek()**.

SEE ALSO: The system request I$Seek in the "OS-9/68000 Operating System Technical Manual" and the function **fseek()**.

malloc() **Allocate Memory from an Arena** **malloc()**

Memory Management

SYNOPSIS: **char *malloc(size)**
unsigned size;

DESCRIPTION: **malloc()** returns a pointer to a block of memory of length "size" bytes. The pointer is suitably aligned for storage of data of any type.

malloc() maintains an amount of memory called an "arena" from which it grants memory requests. **malloc()** will search its arena for a block of free memory large enough for the request and, in the process, coalesce adjacent blocks of free space returned by the **free()** function. If insufficient memory is available in the arena, **malloc()** calls **ebrk()** to get more memory from the system.

malloc() returns **NULL** (0) if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of an assigned block.

CAVEATS: Utter chaos will ensue if the space assigned by **malloc()** is overrun or if **free()** is handed a value not assigned by **malloc()**.

SEE ALSO: **free(), ebrk(), ibrk(), sbrk().**

mknod() **Create a Directory** **mknod()**

UNIX System

SYNOPSIS:

```
#include <modes.h>

mknod(name,perm)
char *name;
short perm;
```

DESCRIPTION: mknod() is used to create a directory file. The name of the directory is given by the pointer "name", and the desired access permissions of the directory are given by "perm".

The function returns zero if the directory was successfully created. If the creation failed, -1 is returned and the appropriate error code is placed in the global variable "errno".

CAVEATS: mknod() does not make UNIX-style "special files" as there is no such thing on OS-9.

SEE ALSO: The I$MakDir system request in the "OS-9/68000 Operating System Technical Manual".

mktemp() Create a File with a Unique Name mktemp()

Miscellaneous

SYNOPSIS:

```
char *mktemp(name)
char *name;
```

DESCRIPTION: mktemp() can be used to ensure that the name of a temporary file is unique in the system and does not clash with any other file name.

A pointer to a template "name" is passed to the function. The template string should look like a filename with six trailing "X"s. mktemp() will replace the "X"s with a letter and the current process id. The letter will be chosen so that the resulting name will not conflict with an existing file.

mktemp() returns a pointer to the template or NULL if more than 26 mktemp() files exist.

modf() **Return Parts of a Real Number** **modf()**

Mathematical

SYNOPSIS:
```
double modf(value,iptr)
double value,*iptr;
```

DESCRIPTION: modf() returns the signed fractional part of "value" and stores the integral part in the double pointed to by "iptr".

modlink() Link to a Memory Module modlink()

OS-9 System

SYNOPSIS:

```
#include <module.h>

mod_exec *modlink(modname,typelang)
char *modname;
short typelang;
```

DESCRIPTION: modlink() will search the module directory for a module with the same name as that pointed to by "modname" and link to it, provided that "typelang" match the respective value of Type/Language in the module. If the module is found, the link count for the module is incremented by one. If any module type or language is desired, zero can be given to indicate such.

The header file <module.h> contains a structure appropriate for accessing the elements of system-defined memory modules.

If the link is successful, modlink returns a pointer to the module. If the link fails, -1 is returned and the appropriate error code is placed in the global variable "errno".

SEE ALSO: The F$Link system request in the "OS-9/68000 Operating System Technical Manual" and the function modload().

modload() Load and Link to a Memory Module modload()

OS-9 System

SYNOPSIS:

```
#include <module.h>

mod_exec *modload(modname,accessmode)
char *modname;
short accessmode;
```

DESCRIPTION: modload() will search the module directory for a module with the same name as that pointed to by "modname" and link to it. "Accessmode" is the mode with which to open the file to load.

If the module is not in the module directory, "modname" is considered a path and all modules in the file are loaded. A link is made to the first module loaded from the file and a pointer to it is returned. If any module accessmode is acceptable, zero can be given for "accessmode" to indicate such.

The <module.h> header file contains a structure appropriate for accessing the elements of system-defined memory modules.

If the load is successful, modload() returns a pointer to the module. If the load fails, -1 is returned and the appropriate error code is placed in the global variable "errno".

SEE ALSO: The F$Load system request in the "OS-9/68000 Operating System Technical Manual" and the function modlink().

munlink() Unlink from a Module **munlink()**

OS-9 System

SYNOPSIS:

```
#include <module.h>

munlink(module)
mod_exec *module;
```

DESCRIPTION: munlink() informs the system that the module pointed to by "module" is no longer required by the process. The module's link count is decremented, and the module is removed from memory if the link count reaches zero. The pointer "module" must have been a pointer returned by modload() or modlink().

SEE ALSO: The F$Unlink system request in the "OS-9/68000 Operating System Technical Manual" and the functions modlink() and modload().

munload() Unload a Module munload()

OS-9 System

SYNOPSIS:

```
#include <module.h>

munload(name,type)
char *name;
short type;
```

DESCRIPTION: munload() informs the system that the module whose name is pointed to by "name" with a type of "type" is no longer required by the process. The module's link count is decremented, and is removed from memory when the link count reaches zero. A "type" of zero means any type.

CAVEATS: This function differs from munlink() in that it unlinks by module name rather than module pointer. This is the only way to unlink a module on Level Two systems. Attempting to unlink a module by performing a link to a module by name to determine its address and then unlinking it twice will not work because the first unlink removes the module from the process' address space.

SEE ALSO: The F$UnLoad system request in the "OS-9/68000 Operating System Technical Manual" and the functions munlink(), modlink() and modload().

open() Open a File **open()**

UNIX System

SYNOPSIS:

```
#include <modes.h>

int open(name,mode)
char *name;
short mode;
```

DESCRIPTION: open() opens an existing file of name "name". The access mode desired is specified in the argument "mode". The values for "mode" are defined in the header file <modes.h>.

open() returns a path number used to identify the file when I/O is performed. If the open fails, -1 is returned and the appropriate error code is placed in the global variable "errno".

Reads and writes to the file start at the beginning of the file. The file pointer can be moved at any time by the lseek() function.

SEE ALSO: The I$Open system request in the "OS-9/68000 Operating System Technical Manual" and the functions creat(), create(), read(), write(), dup(), and close().

les valeurs de "mode"
attention: non conformes au
standard UNIX

opendir() Open the Directory opendir()

Miscellaneous

SYNOPSIS:

```
#include <dir.h>
dir *opendir (filename)
char *filename
```

DESCRIPTION: Opendir() opens the specified directory and associates a directory stream with it. Opendir() returns a pointer to be used to identify the directory stream in subsequent operations. The pointer NULL is returned if **filename** can not be accessed or if it can not "malloc" enough memory to hold the whole thing.

SEE ALSO: **closedir()**, **readdir()**, **rewinddir()**, **seekdir()** and **telldir()**.

os9exec() **OS-9 System Call Processing** **os9exec()**

OS-9 System

SYNOPSIS:

```
os9exec(procfunc, modname, argv, envp, stacksize, priority, pathcnt)
int (*procfunc)();
char *modname;
char **argv;
char **envp;
unsigned datasize;
short priority, pthcnt;
```

DESCRIPTION: The OS-9 F$Fork system call passes information to a new process as binary data specified by a pointer and size. It is up to the forked process to interpret the data.

The **os9fork()** and **chain()** functions are actually C-level hooks to the system call's themselves. By themselves, they provide no real argument information. Unfortunately, this leaves the interpretation of a raw argument string up to the cstart portion of the C program. Problems can occur when handling quotes and control characters.

The **os9exec()** function provides more precise control over the argument strings as well as supplying the environment variables inherited from the parent process. The **os9exec()** function closely simulates the UNIX **execve()** system call.

procfunc is a pointer to a function that creates a new process. This function is normally **os9fork**, **chain()**, **os9forkc** or **chainc()**.

modname is a pointer to a string naming the new primary module.

argv is the argument pointer list that the new process is to receive. By convention, argv[0] is the name of the new module and the end of the argv list is marked by a NULL pointer.

stacksize is the additional memory (in bytes) to allocate to the new proces' stack memory. A value of zero indicates the default stack size required by the process.

priority is the priority value at which the new process is to run. If the value is zero, the new process is given the priority of the calling process.

OS-9 System
(continued)

pathcnt is given only if the procfunc value indicates os9forkc() or chainc(). This value is the number of open paths to pass to the new process. Normally, three is given for this argument, causing the three standard paths to be passed. A value of zero passes NO open paths.

NOTE: Any program that executes a system command should use the os9exec() interface. The cstart module can handle parameter strings passed by the os9fork() and chain() functions, but no environment is available and the argv pointer list is separated by white space.

EXAMPLE:

```
        extern int os9forkc();
        extern char **environ;

        char *argblk[] = {
                "rename",
                "-x",
                "oldname",
                "newnwme",
                0,
        };

        main ()
        {
                .
                .
                .
                if ((pid = os9exec(os9fork,argblk[0],argblk,
                    environ,0,0,3)) > 0) wait(0);
                else printf ("can't fork %s",argblk[0]);
                .
                .
                .
        }
```

Above is an example call to os9exec(). The "argblk" array contains pointers to the arguments in the order that the new program is to receive them. The new process will receive a copy of the arguments, not the addresses of the arguments. This exemple passes the current environment by naming the global variable "environ".

os9fork() Create a Process os9fork()
os9forkc() os9forkc()

OS-9 System

SYNOPSIS:

```
os9fork(modname,parmsize,parmptr,type,lang,datasize,prior)
char *modname,*parmptr;
int parmsize,datasize;
short type,lang,prior;

os9forkc(modname,parmsize,parmptr,type,lang,datasize,
        prior,pathcnt)
char *modname,*parmptr;
int parmsize,datasize;
short type,lang,prior,pathcnt;
```

DESCRIPTION: os9fork() will create a process that will run concurrently with the calling process. When the new process terminates, its exit status is available to the forking (parent) process. The new process will "inherit" the standard paths (path numbers 0, 1 and 2).

The argument "modname" is a pointer to a null-terminated module name. The argument "parmptr" is a pointer to a null-terminated string to be passed to the new process; "parmsize" is the strlen() of "parmptr". The arguments "type" and "lang" specify the desired type and language of the module; these can be given as zero to indicate any type or language. The argument "datasize" gives extra memory to the new program. If no extra is required, this value can be zero. Finally, the argument "prior" is the new priority at which to run the program; zero can be given if no priority change is desired.

If the fork is unsuccessful, os9fork() returns -1 and the appropriate error code is placed in the global variable "errno". If the fork is successful, the process id number of the new process is returned.

os9forkc() is the same as os9fork() with a pathcnt argument. pathcnt is the number of open paths for the new process to inherit.

os9exec() is the preferred method by which to fork C programs. See the discussion of os9exec() for more details.

CAVEATS: Beware of forking to a system module. Usually, it only makes sense to fork to a program of type "object module".

SEE ALSO: The discussion of F$Chain and F$Fork in the "OS-9/68000 Operating System Technical Manual" and the discussion of chain() and os9exec().

pause() Wait for Signal pause()

UNIX System

SYNOPSIS: pause()

DESCRIPTION: pause() may be used to suspend a task until a signal is received. pause() always returns -1 unless the signal received caused the process to terminate, in which case pause() never returns.

SEE ALSO: The F$Sleep system request in the "OS-9/68000 Operating System Technical Manual" and the functions sleep() and kill().

pffinit() Initialize for Float Output (obsolete) pffinit()

Standard I/O

SYNOPSIS: pffinit()

DESCRIPTION: This function was used on the 6809 to allow printf to perform float and double conversions. A dummy function named pffinit() exists here for 6809/68000 portability.

pflinit() **Initialize for Longs Output (obsolete)** **pflinit()**

Standard I/O

SYNOPSIS: pflinit()

DESCRIPTION: This function was used on the 6809 to allow printf to perform long int conversions. A dummy function named pflinit() exists here for 6809/68000 portability.

pow() **Power Function** **pow()**

Mathematical

SYNOPSIS:
```
#include <math.h>

double pow(x, y)
double x, y;
```

DESCRIPTION: pow() returns "x" raised to the power "y". The values of "x" and "y" may not both be zero. If "x" is not positive, "y" must be an integer.

prerr() **Print error Message** **prerr()**

UNIX System

```
SYNOPSIS:  prerr(path,errnum)
           int path;
           short errnum;
```

DESCRIPTION: prerr() prints an error message on the standard output path. If "path" is zero the message corresponding to the error number "errnum" is printed as "ERROR #mmm.nnn", mmm being the high byte of the error number, nnn being the lower. If "path" is non-zero, that path is assumed to be a file containing error message text which is printed along with the error number.

SEE ALSO: The F$PErr system request in the "OS-9/68000 Operating System Technical Manual".

printf() Formatted Output **printf()**

Standard I/O

SYNOPSIS: #include <stdio.h>

```
int printf(control [,arg0[,arg1...]])
char *control;
```

DESCRIPTION: printf(), fprintf(), and sprintf() are C standard library functions that perform formatted output. Each of these functions converts, formats, and prints the "args" (if any) as indicated by the "control" string. printf() places its output on standard output.

The control string determines the format, type, and number of the following arguments expected by the function. If the control string does not match the arguments correctly, the results are unpredictable.

The control string may contain characters to be copied directly to the output, and/or format conversion specifications. Each format specification causes the function to take the next successive argument for output conversion.

A format specification consists of a "%" character followed by (in this order):

1. An optional minus sign ("-") that means the field will be left justified.

2. An optional string of digits indicating the field width required. The field will be at least this wide and may be wider if the conversion requires it. The field will be right justified (padded on the left) unless the minus sign is present (padded on the right). The default padding character is a space. If the digit string starts with a zero, the padding character will be "0".

3. An optional period (".") and a digit string (the precision). For floating point arguments this indicates the number of digits to follow the decimal point on conversion. For strings; the maximum number of characters of the string argument to be printed.

printf() (continued)

4. An optional character "l" indicating that the following "d", "x" or "o" is the specification for a long integer argument. Because, in this compiler, the types long and int are synonymous, the "l" has no effect. It appears for ease of porting programs to and from other machines.

5. A conversion character indicates the argument type and the desired conversion. The recognized conversion characters are:

d,o,x,X	The argument is an integer. The conversion is to either a signed integer, octal, or hexadecimal, respectively. The ranges for the types are: signed integer, -2,147,433,648 to 2,147,483,647; octal, 0 to 037777777777; and hex, 0 to 0xffffffff. The conversion "X" prints the alphabetic letters of a hex number in upper case rather than lower case.
u	The argument is an integer. The conversion is to an unsigned integer in the range 0 to 4,294,967,295.
f	The argument is a double. The form of the conversion is "[-]nnn.nnn", where the digits after the decimal point are specified as above. If not given, the precision defaults to six digits. If the precision is zero, no decimal point or following digits are printed.
e,E	The argument is a double. The form of the conversion is "[-]n.nnne(+/-)nnn". One digit appears before the decimal point. The precision gives the digits following the decimal point. When the precision is missing, six digits are produced. If the precision is zero, no decimal point appears. The conversion E will produce a number using "E" instead of "e" to introduce the exponent. The exponent always contains 3 digits. The resulting value is rounded before printing.

printf() (continued)

g,G	The argument is a double. The form of the conversion is style "e" or "f" depending on the value resulting from the conversion. "e" format will be used only if the exponent resulting from the conversion is less than -4 or greater than the precision. If the precision is missing, it is assumed to be six. Trailing zeroes are removed from the result. A decimal point appears only if followed by a digit. The resulting value is rounded before printing.
c	The argument is printed as a character.
s	The argument is a pointer to a string. Characters from the string are printed up to a null byte, or until the number of characters indicated by the precision have been printed. If the precision is zero or missing, the characters are not counted.
% (or any unrecognized character)	No argument corresponding; the character is printed.

CAVEATS: Most errors with **printf()** are caused by the arguments being passed not corresponding in type and number with the control string, which will always cause unpredictable results. Also, passing a NULL pointer (0) as the pointer for the %s conversion is even more unpredictable.

SEE ALSO: **fprintf()**, **sprintf()**, and **scanf()**.

EXAMPLES:

```
printf("Value %d(dec), %x(hex), %o(octal)\n",val,val,val);

printf(s,x,y,z);             /* s points to control string */

printf("%-16.16g",fltans);          /* 16-char general float
                                       field */
```

putc() Put Next Character to File putc()

Standard I/O

SYNOPSIS:

```
#include <stdio.h>

int putc(c,fp)
int c;
FILE *fp;
```

DESCRIPTION: putc() writes the character "c" to the file pointed to by "fp". putc() returns a -1 if the function fails. If successful, it returns the character passed.

Under OS-9, there is a choice of low-level service requests to use when reading from a file.

write()	will output characters up to a specified bytecount in "raw" mode. No conversion will take place on the output stream and the characters will appear in the file (or device) exactly as written.
writeln()	honors the various mappings of characters associated with a Serial Character device such as a terminal and in any case will return to the caller as soon as a carriage return is written. Normal output conversion consists of possible conversion of carriage return to carriage return/linefeed, expansion of tab to spaces, handling of screen line count pause, etc.

In the vast majority of cases, it is preferable to use writeln() for accessing Serial Character devices and write() for any other file output. putc() uses this strategy. As file output is through putc(), so do all the other library output functions.

The choice is made when the first call to putc() is made after the file is opened. The system is consulted for the status of the file. A flag bit is set in the file structure accordingly.

The choice may be forced by the programmer by setting the relevant bit before a call to putc(). The flag bits are defined in the <stdio.h> header file as _RBF and _SCF.

The following method is used to set the flag bits. Assuming that the file pointer (as returned by fopen()) is fp, the following will force the use of writeln() and write() on output, respectively:

```
fp->_flag |= _SCF;

fp->_flag |= _RBF;
```

putc() (continued)

This trick may be played on the standard streams, "stdout" and "stderr", without the need for calling fopen() but must be done before any output is performed on the stream.

When it is desired to output characters one byte at a time without buffering, use this technique:

```
fp->_flag |= _UNBUF;
```

This will cause data to be written to the file one byte at a time.

If a file is open for update (fopen() action "r+" or "w+") the program is required to do an **fseek()** before changing from **getc()** to **putc()** or **putc()** to **getc()**, even if no effective file position change occurs. This is done to cause the buffer to be flushed (or filled) so input or output can proceed in the opposite direction.

CAVEATS: It is not a good idea to call **read()** or **write()** on a path that is buffered since the buffered data and the low-level I/O may not occur in the correct order.

NOTE: That due to the buffering, RBF record-locking is ineffective. If an application requires the OS-9 record-locking facilities, low-level I/O calls and program buffering is required.

SEE ALSO: **fopen()**, **getc()**.

putchar() Put Next Character to Standard Output **putchar()**

Standard I/O

SYNOPSIS:

```
#include <stdio.h>

int putchar(c)
char c;
```

DESCRIPTION: The macro putchar(c) is equivalent to: putc(c,stdout). The return value is the character passed, or -1 if an error occurred.

SEE ALSO: putc().

puts() **Output a String to a File** **puts()**

Standard I/O

SYNOPSIS:
```
puts(ptr)
char *ptr;
```

DESCRIPTION: puts() copies the null terminated string pointed to by "ptr" onto the file "stdout". With the exception that a "\n" is written out after the string, the effect is exactly that of:

```
fputs(ptr,stdout)
```

The string terminating zero byte is not copied. puts() returns its first argument, or NULL (0) if an error occurs.

CAVEATS: The inconsistency of the new line being appended by puts() and not by fputs() is dictated by history and the desire for compatibility.

SEE ALSO: puts(), gets(), fgets().

putw() Put a Word to a File putw()

Standard I/O

SYNOPSIS:

```
#include <stdio.h>

int putw(w, fp)
int w;
FILE *fp;
```

DESCRIPTION: putw() writes the integer "w" to the file pointed to by "fp". putw() neither assumes nor causes any special alignment in the file. On success, putw() returns the value it has written, otherwise it returns -1.

CAVEATS: putw(), like getw(), is machine-dependent because the size of the integer it outputs varies with the integer size of the machine on which it resides. This compiler defines int values as 4 byte quantities.

qsort() **Quick Sort** **qsort()**

Miscellaneous

SYNOPSIS:
```
qsort(base, n, size, compfunc)
char *base;
int n,size;
int (*compfunc)();      /* pointer to func returning int */
```

DESCRIPTION: qsort() implements the quick-sort algorithm for sorting an arbitrary array of items.

The argument "base" is the address of the array of "n" items of size "size". The argument "compfunc" is a pointer to a comparison function supplied by the user (or usually just **strcmp()**). The comparison function will be called by **qsort()** with two pointers to items in the array to be compared and should return an integer which is less than, equal to, or greater than 0, where, respectively, the first item is less than, equal to, or greater than the second.

read() **Read bytes from a Path** **read()**

UNIX System

SYNOPSIS:

```
int read(path, buffer, count)
int path;
char *buffer;
unsigned count;
```

DESCRIPTION: **read()** reads bytes from a path. The path number "pn" is an integer which is one of the standard path numbers 0, 1, or 2, or a path number returned from a successful call to **open()**, **creat()**, **create()**, or **dup()**. "buffer" is a pointer to space with at least "count" bytes of memory into which **read()** will put the data read from the path.

It is guaranteed that at most "count" bytes will be read, but often less will be, either because the path serves a terminal and input stops at the end of a line or the end-of-file has been reached.

readln() causes "line-editing" such as echoing to take place and returns once a "\n" is read in the input or the number of bytes requested has been read. **readln()** is the preferred call for reading from the terminal.

read() essentially does a raw read, that is, the read is performed without translation of characters. The characters are passed to the program as read.

read() and **readln()** return the number of bytes actually read (0 indicating the end of the file). If an error occurs, they return -1 and the appropriate error code is placed in the global variable "errno".

CAVEATS: Notice that end-of-file is returned as zero bytes being read, not an error indication.

SEE ALSO: The I$Read and I$ReadLn system requests in the "OS-9/68000 Operating System Technical Manual" and the **readln()** function.

readdir() **Returns a Pointer** **readdir()**

Miscellaneous

SYNOPSIS:
```
#include <dir.h>
struct direct *readdir(dirp)
dir *dirp
```

DESCRIPTION: readdir() returns a pointer to a structure containing the next directory entry. It returns NULL upon reaching the end of the directory or detecting an invalid seekdir() operation.

SEE ALSO: closedir(), opendir(), rewinddir(), seekdir() and telldir().

readln() Read bytes from a Path readln()

OS-9 System

SYNOPSIS:

```
int readln(path, buffer, count)
int path;
char *buffer;
unsigned count;
```

DESCRIPTION: See the discussion of read() for complete details on readln().

SEE ALSO: The I$Read and I$ReadLn system requests in the "OS-9/68000 Operating System Technical Manual" and the readln() function.

rewind() **Return File Pointer to Zero** **rewind()**

Standard I/O

SYNOPSIS:
```
#include <stdio.h>

int rewind(fp)
FILE *fp;
```

DESCRIPTION: rewind() is equivalent to:

```
fseek(fp,0,0);
```

Zero is returned if the rewind is successful, otherwise -1 is returned.

SEE ALSO: fseek().

r() Resets the Position of the directory stream rewinddir()

Miscellaneous

```
:  #include <dir.h>
   rewinddir(dirp)
   dir *dirp;
```

ION: rewindir() resets the position of the named directory
o the beginning of the directory.

: closedir(), opendir(), readdir(), seekdir() and telldir().

rindex() **Search for Character in String** **rindex()**

String Handling

SYNOPSIS:

```
char *rindex(ptr,ch)
char *ptr;
char ch;
```

DESCRIPTION: rindex() returns a pointer to the last occurrence of the character "ch" in the string pointed to by "ptr". If the character is not found, the function returns a NULL (0).

CAVEATS: Note that "ch" is a character, not a pointer to a character:

```
func()
{
     char *ptr,*string;

     .
     .
     .

     if((ptr = rindex(string,'.')) {
             process(ptr);
     } else {
             printf("No '.' found!\n");
     }
}
```

The above example looks for a '.' (period) in the string "string" and sets "ptr" to point to it. The searching is started from the end of the string and progresses to the front of the string.

SEE ALSO: index().

sbrk() Extend Data Memory Segment **sbrk()**

Memory Management

SYNOPSIS: char *sbrk(size)
unsigned size;

DESCRIPTION: sbrk() is a function to allocate memory from the top of the data area upwards.

```
|                |  ^
|   Sbrk Area    |  |
|                |  ----- Grows towards higher address
>----------------<
|                |
|  Stack Area    |
|                |

|                |
|   Ibrk Area    |
|                |
>----------------<
|                |
|    Program     |
|   Data Area    |
|                |
'----------------'
```

sbrk() grants memory requests by calling the OS-9 F$Mem service request. This method resizes the data area to a larger size; the new memory granted is contiguous with the end of the previous data memory.

In Level One systems, this call is certain to fail quickly, because it is easy to keep growing in size until the data area touches other allocated memory. At this point, it is impossible to increase in size and an error is returned. A program may be able to increase its data size only 20k, even if there is 200k available elsewhere.

To gain the most utility of the 68000 addressing space, use the ebrk() function which returns pointers to memory no matter where it is located in the system.

SEE ALSO: The F$Mem system request in the "OS-9/68000 Operating System Technical Manual" and the functions ibrk() and ebrk().

scanf() **Input Strings Conversion** **scanf()**

Standard I/O

SYNOPSIS: `#include <stdio.h>`

```
scanf(control [,arg...])
char *control;
```

DESCRIPTION: scanf() performs conversions from the file pointer "stdin" (which is equivalent to fscanf(stdin,control[,arg...]).

Each form of the function (scanf(), fscanf(), and sscanf()) expects a control string, similar to printf, containing conversion specifications, and zero or more pointers to objects into which the converted values are stored.

The functions return EOF (-1) on end of input or error, or a count of the items successfully matched. If the count does not indicate the expected number of items, a match problem occurred.

The control string may contain three types of fields:

1. Spaces, tab characters, or "\n" which match any of the three in the input.
2. Characters (not among the above nor "%") which must match characters in the input.
3. A "%" followed by an optional "*" denoting suppression of assignment, an optional field width maximum, and a conversion character indicating the type expected.

A conversion character controls the conversion to be applied to the next field and indicates the type of the corresponding pointer argument. A field consists of consecutive non-space characters and ends when either an input character is inappropriate for the conversion or a specified field width is exhausted. When one field is finished, white space characters are passed over until the next field is found.

The following conversion characters are recognized:

d	A decimal string is expected on input and the argument must be a pointer to an integer.
o	An octal string; the corresponding argument should point to an integer.

scanf() (continued)

x	A hexadecimal string for conversion; the argument must be a pointer to an integer.
s	A string of non-space characters is expected and will be copied to the buffer pointed to by the corresponding argument with a null byte appended. The caller must ensure the buffer is large enough for the string. The input string is terminated by a space, tab, or newline ('\n').
c	A character is expected and is copied into the byte pointed to by the argument. The white space skipping is suppressed for this conversion. If a field width is given, the argument is assumed to point to a character array and the number of characters indicated is copied to it. To ensure that the next non-white space character is read, use "%1s" with an argument that points to at least FOUR bytes.
e,f	A floating point representation is expected; the argument must be a pointer to a float. Any of the usual ways of writing floating point numbers are recognized.
[	This denotes the start of a set of match characters, the inclusion or exclusion of which delimits the input field. The white space skipping is suppressed. The corresponding argument should be a pointer to a character array. If the first character in the match string is not a "^", characters are copied from the input as long as they can be found in the match string. If the first character is a "^", then copying continues while the characters can not be found in the match string. The match string is delimited by a "]".
D,O,X	Similar to d,o,x above, but the argument is assumed to point to a long. In this compiler, long and int are synonymous.
E,F	Similar to e and f above, but the argument is assumed to point to a double.
%	A match for "%" is sought; no conversion takes place.

scanf() (continued)

The conversion characters "d", "u", "o", and "x" may be preceded by "l" or "h" to indicate that a pointer to a long or short rather than to an int is in the argument list. Each of the functions returns a count of the number of fields successfully scanned. scanf() will terminate at the end of the control string, when end-of-file is encountered, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

CAVEATS: The returned count of matches/assignments does not include character matches and assignments suppressed by "*". The arguments must all be pointers. It is a common error to call scanf() with the value of an item rather that a pointer to it. Also, the '\n' of an input line must be explicitly matched.

SEE ALSO: fscanf(),sscanf().

seekdir() Sets the Position of the Next Readdir seekdir()

Miscellaneous

SYNOPSIS:

```
#include <dir.h>
seekdir(dirp, loc)
dir *dirp;
long loc;
```

DESCRIPTION: seekdir() sets the position of the next readdir() operation on the directory stream. The new position reverts to the one associated with the directory stream when the telldir operation was performed.

Values returned by telldir are valid only for the lifetime of the associated dir pointer. If the directory is closed and then reopened, the telldir() value may be invalidated. It is safe to use a previous telldir() value immediately after a call to opendir() and before any calls to readdir().

SEE ALSO: closedir(), opendir(), readdir(), rewinddir() and telldir().

setbuf() **Fix File Buffer** **setbuf()**

Standard I/O

SYNOPSIS:

```
#include <stdio.h>

setbuf(fp,buffer)
FILE *fp;
char *buffer;
```

DESCRIPTION: When the first character is written to or read from a file (via getc() and putc()) after it has been opened by fopen(), a buffer is obtained from the system (if required) and assigned to the file. setbuf() may be used to forestall this by assigning a user buffer to the file.

setbuf() must be used after the file has been opened and before any I/O has taken place.

The buffer must be of sufficient size. A manifest constant, BUFSIZ, is defined in the <stdio.h> header file that is normally assigned as the buffer size.

If the "buffer" argument is NULL (0), the file becomes unbuffered and characters are read and written singly.

SEE ALSO: getc(), putc(), and fopen().

setime() Set System Time **setime()**

UNIX System

SYNOPSIS: `#include <time.h>`

```
setime(timebuf)
struct sgtbuf *timebuf;
```

DESCRIPTION: setime() sets the system time from the time buffer pointed to by "timebuf". The time units are defined in the <time.h> header file.

If successful, setime() returns zero. Otherwise, -1 is returned and the appropriate error code is placed in the variable "errno".

SEE ALSO: getime().

setjmp() **Non-local Goto** **setjmp()**

Miscellaneous

SYNOPSIS: **#include <setjmp.h>**

```
setjmp(env)
jmp_buf env;
```

DESCRIPTION: The **setjmp()** and **longjmp()** functions provide a way to perform gotos between functions in C. See the discussion of **longjmp()** for full details on **setjmp()**.

EE ALSO: **longjmp()**.

setpr() Set Process Priority setpr()

UNIX System

SYNOPSIS:

```
setpr(pid,prior)
int pid;
short prior;
```

DESCRIPTION: setpr() sets the process indicated by "pid" to have a priority of "prior". The lowest priority is zero, the highest is 65535.

setpr() will return -1 if an error occurs such as the process not having the same user id as the caller. If an error occurs, the appropriate error code is placed in the global variable "errno".

setstat() **Set File Status** **setstat()**

OS-9 System

SYNOPSIS:

```
#include <sgstat.h>

setstat(code,path,buffer)          /* code == 0 */
int code,path;
char *buffer;

setstat(code,path,size)            /* code == 2 */
int code,path;
long size;
```

DESCRIPTION: The setstat() function is used to set the path options or the file size of the file open on "path".

If "code" is zero, the "buffer" is copied to the path descriptor options section. The header file <sgstat.h> contains the definitions for the path options. If "code" is 2, "size" should be an int specifying the new file size.

If an error occurs, both forms of the call will return -1 and place the appropriate error code in the global variable "errno".

CAVEATS: This call exists for 6809 portability. The "_ss" functions are the preferred and more palatable versions of these function calls.

SEE ALSO: The I$SetStt service request in the "OS-9/68000 Operating System Technical Manual", the function getstat(), and any function name beginning with "_ss".

setuid() **Set User ID** **setuid()**

UNIX System

SYNOPSIS: int setuid(uid)
int uid;

DESCRIPTION: setuid() sets the group/user id of the process to "uid". This call only works if the current group/user id is the super user (user id 0).

If the call fails, -1 is returned and the appropriate error code is placed in the global variable "errno".

SEE ALSO: getuid().

sin() **Sine Function** **sin()**

Mathematical

SYNOPSIS:

```
#include <math.h>

double sin(x)
double x;
```

DESCRIPTION: sin() returns the sin of "x", which is in radians.

sleep() Suspend Execution for a Time sleep()

UNIX System

SYNOPSIS:

```
sleep(seconds)
unsigned seconds;
```

DESCRIPTION: sleep() suspends the calling process for the specified time. A sleep time of zero seconds sleeps for 1 clock tick. sleep() returns the number of ticks remaining to be slept if awakened prematurely by a signal.

SEE ALSO: The F$Sleep system request in the "OS-9/68000 Operating System Technical Manual" and the function tsleep().

sprintf() **Formatted Output** **sprintf()**

Standard I/O

SYNOPSIS: int sprintf(buffer, control [,arg0[,arg1...]])
char *buffer;
char *control;

DESCRIPTION: printf(), fprintf(), and sprintf() are C standard library functions that perform formatted output. Each of these functions converts, formats, and prints the "args" (if any) as indicated by the "control" string.

sprintf() places its output into the array pointed to by "buffer"; the string is terminated by a null byte. The function returns the number of characters placed in the buffer, not including the null byte.

The control string determines the format, type, and number of the following arguments expected by the function. If the control string does not match the arguments correctly, the results are unpredictable. See the discussion of printf() for details on the control string.

SEE ALSO: printf(), fprintf().

sqrt() Square Root Function sqrt()

Mathematical

SYNOPSIS:
```
#include <math.h>

double sqrt(x)
double x;
```

DESCRIPTION: sqrt() returns the square root of "x". "x" must be positive.

sscanf() Input Strings Conversion sscanf()

Standard I/O

SYNOPSIS:

```
#include <stdio.h>

sscanf(string,control [,arg...])
char *string;
char *control;
```

DESCRIPTION: **sscanf()** performs conversions from the input string pointed to by "string". **sscanf()** expects a control string, similar to printf, containing conversion specifications, and zero or more pointers to objects into which the converted values are stored.

The function will return EOF (-1) on end of input or error, or a count of the items successfully matched. If the count does not indicate the expected number of items, a match problem occurred.

The control string is described in detail in the discussion of **scanf()**.

SEE ALSO: scanf(), fscanf(), printf().

stacksiz() **Obtain Size of Stack Used** **stacksiz()**

Miscellaneous

SYNOPSIS: int stacksiz()

DESCRIPTION: If the stack checking code is in effect, a call to stacksiz() will return the maximum number of bytes of stack used at the time of the call. This function can be used to determine the stack size required by a program.

SEE ALSO: freemem().

strcat() **String Catenation** **strcat()**

String Handling

SYNOPSIS:
```
char *strcat(s1,s2)
char *s1,*s2;
```

DESCRIPTION: **strcat()** appends a copy of the string pointed to be "s2" to the end of the string pointed to by "s1". Null-byte terminated strings are assumed. **strcat()** returns its first argument.

CAVEATS: The calling routine is responsible for ensuring adequate space to append "s2" to "s1".

strcmp() String Comparison strcmp()

String Handling

SYNOPSIS:

```
char *strcmp(s1,s2)
char *s1,*s2;
```

DESCRIPTION: strcmp() compares the strings pointed to by "s1" and "s2" for lexicographic order and returns an integer less than, equal to, or greater than zero, where, respectively, "s1" is less than, equal to, or greater than "s2". Null-byte terminated strings are assumed.

strcpy() String Copy strcpy()

String Handling

SYNOPSIS:
```
char *strcpy(s1,s2)
char *s1,*s2;
```

DESCRIPTION: strncpy() copies characters from "s2" to the space pointed to by "s1". Null-byte terminated strings are assumed. If "s2" is too long, "s1" will not be null-terminated.

strncpy() returns its first argument.

CAVEATS: The function assumes that there is adequate space pointed to by "s1" to do the copy. The calling routine is responsible for ensuring that the space is adequate.

strhcpy() Copy Old OS-9 Strings **strhcpy()**

String Handling

SYNOPSIS:

```
char *strhcpy(s1,s2)
char *s1,*s2;
```

DESCRIPTION: strhcpy() makes a copy of the string pointed to by "s2" in the string pointed to by "s1". The "s2" string is assumed to have the high-order bit set on the character byte indicating the last character. The function copies the bytes from "s2" until the terminator to "s1" removes the high-bit from the terminator character. It then appends a null byte to the output string.

This function is used primarily for copying directory names from OS-9 6809-style RBF disks.

The function returns its first argument.

CAVEATS: strhcpy() assumes there is adequate space pointed to by "s1" to do the copy. The calling routine is responsible for verifying the space.

strlen() Determine String Length strlen()

String Handling

SYNOPSIS:

```
int strlen(s)
char *s;
```

DESCRIPTION: strlen() returns the number of non-null characters in the string pointed to by "s". The function stops when the first null byte is encountered.

NB: The returned length does not include the terminated NUL character

strncat() String Catenation strncat()

String Handling

SYNOPSIS:

```
char *strncat(s1,s2,count)
char *s1,*s2;
int count;
```

DESCRIPTION: strncat() appends a copy of the string pointed to by "s2" to the end of the string pointed to by "s1". The function copies, at most, "count" characters. Null-byte terminated strings are assumed. strncat() returns its first argument.

CAVEATS: The function assumes there is room to copy "s2" at the end of "s1". Space needs to be properly allocated by the calling routine.

strncmp() **String Comparison** **strncmp()**

String Handling

SYNOPSIS:
```
char *strncmp(s1,s2,count)
char *s1,*s2;
int count;
```

DESCRIPTION: strncmp() compares the strings pointed to by "s1" and "s2" for lexicographic order and returns an integer less than, equal to, or greater than zero, where, respectively, "s1" is less than, equal to, or greater than "s2". The function compares, at most, "count" characters. Null-byte terminated strings are assumed.

strncpy() String Copy **strncpy()**

String Handling

SYNOPSIS:

```
char *strncpy(s1,s2,count)
char *s1,*s2;
int count;
```

DESCRIPTION: strncpy() copies characters from "s2" to the space pointed to by "s1", the number of which is determined by "count".

If the string "s2" is too short, "s1" will be padded with null bytes to make up the length difference. If "s2" is too long, "s1" will not be null-terminated.

Null-byte terminated strings are assumed. strncpy() returns its first argument.

CAVEATS: The function assumes that there is adequate space pointed to by "s1" to do the copy. The calling routine is responsible for ensuring that the space is adequate.

system() Shell Command Execution system()

Miscellaneous

SYNOPSIS:

```
system(string)
char *string;
```

DESCRIPTION: system() passes its argument to "shell", which executes it as a command line. The calling process is suspended until the shell command is completed. The maximum length of "string" is 80 characters. If a longer string is needed, use os9fork(). system() returns 0 if the command was successful and -1 if the command failed.

SEE ALSO: The functions os9fork() and wait().

tan() Tangent Function tan()

Mathematical

SYNOPSIS:

```
#include <math.h>

double tan(x)
double x;
```

DESCRIPTION: tan() returns the tangent of "x", which is in radians.

CAVEATS: tan() may cause a "trapv exception" for values close to PI/2.

telldir() Returns the Current Location **telldir(**

Miscellaneous

SYNOPSIS:
```
#include <dir.h>
long telldir(dirp)
dir *dirp
```

DESCRIPTION: telldir() returns the current location associated with the named directory stream.

SEE ALSO: closedir(), opendir(), readdir(), rewinddir() and seekdir().

Character Conversion

SYNOPSIS:

```
#include <ctype.h>

int toascii(c)
int c;
```

DESCRIPTION: toascii() returns its argument with all bits turned off th[illegible] are not part of a standard ASCII character; it is intended for compatibility with other systems.

SEE ALSO: getc().

Character Conversion

SYNOPSIS: #include <ctype.h>

```
int tolower(c)
int c;
```

DESCRIPTION: tolower() is a genuine function (as opposed to _tolower() which is a macro) that requires a character in the range of -1 to 255. If the argument represents an upper-case letter, the value returned is the corresponding lower-case letter. All other values in the domain are returned unchanged.

SEE ALSO: _tolower().

toupper() Character Translation toupper()

Character Conversion

SYNOPSIS: #include <ctype.h>

```
int toupper(c)
int c;
```

DESCRIPTION: toupper() is a genuine function (as opposed to _toupper() which is a macro) that requires a character in the range of -1 to 255. If the argument represents a lower-case letter, the value returned is the corresponding upper-case letter. All other values in the domain are returned unchanged.

SEE ALSO: _toupper().

tsleep() **Sleep for Specified Interval** **tsleep()**

OS-9 System

SYNOPSIS: **tsleep(svalue)**
unsigned svalue;

DESCRIPTION: tsleep() deactivates the calling process for a specified interval given by "svalue". If "svalue" is zero, the process will sleep indefinitely. If "svalue" is one, the process gives up a time slice. "svalue" is considered a tick count to sleep. If the high bit of "svalue" is set, the remaining 31 bits are considered the number of 256ths of a second to sleep.

SEE ALSO: The F$Sleep system request in the "OS-9/68000 Operating System Technical Manual" and the function sleep().

ungetc() Unget a Character ungetc()

Standard I/O

SYNOPSIS:

```
#include <stdio.h>

int ungetc(c, fp)
char c;
FILE *fp;
```

DESCRIPTION: ungetc() inserts the character "c" into the buffer associated with the file pointed to by "fp". That character will be returned by the next call to getc() on that file. ungetc() returns the character "c" and leaves the file "fp" unchanged.

One character of pushback is guaranteed provided something has been read from the file and the file is actually buffered.

If "c" is EOF (-1), ungetc() does nothing to the buffer and returns EOF.

CAVEATS: fseek() will cause the pushed back character to be forgotten. A read from the file is required before ungetc() can push back a character. The file "stdin", however, need not be read before a call to ungetc(). Only one character can be pushed back onto a file until a getc() is performed.

SEE ALSO: getc(), fseek(), setbuf().

unlink() **Unlink (Delete) a File** **unlink()**

UNIX System

SYNOPSIS:
```
int unlink(name)
char *name;
```

DESCRIPTION: unlink() decrements the link count of the file whose pathname is "name". When all the links to a file have been removed, the space occupied by the file on the disk is freed and the file ceases to exist.

If successful, the function returns 0. If an error occurs, the value -1 is returned and the appropriate error code is placed in the global variable "errno".

CAVEATS: OS-9 does not yet support multiple links to a file. The unlink() function always causes the file to be removed from the disk. Attempting to delete a file that is open by the calling (or another) process results in a record-lock error.

SEE ALSO: The I$Delete system request in the "OS-9/68000 Operating System Technical Manual".

unlinkx() **Unlink (Delete) a File** **unlinkx()**

OS-9 System

SYNOPSIS: **#include <modes.h>**

```
int unlinkx(name,mode)
char *name;
short mode;
```

DESCRIPTION: unlinkx() decrements the link count of the file whose pathname is "name". If the "mode" passed indicates "execution directory", the path is assumed to be based in the current execution directory. The header file <modes.h> defines the legal mode values. When all the links to a file have been removed, the space occupied by the file on the disk is freed and the file ceases to exist.

If successful, the function returns 0. If an error occurs, the value -1 is returned and the appropriate error code is placed in the global variable "errno".

CAVEATS: OS-9 does not yet support multiple links to a file. The unlinkx() function always causes the file to be removed from the disk. Attempting to delete a file that is open by the calling (or another) process results in a record-lock error.

SEE ALSO: The I$Delete system request in the "OS-9/68000 Operating System Technical Manual".

wait() Wait for Process Termination wait()

UNIX System

SYNOPSIS:

```
wait(status)
unsigned *status;

wait(0)
```

DESCRIPTION: wait() is used to suspend the current process until a child process has terminated.

The call returns the process id of the terminating process and places the exit status of that process in the unsigned int pointed to by "status". If "status" is passed as zero, no child status is available to the caller.

The lower 16-bits of the status value will contain the argument of the exit() or _exit() call as executed by the child process, or the signal number if it was interrupted with a signal. A normally terminating C program with no explicit call to exit() in main() will return a status of zero.

wait() will return -1 if there is no child process to wait for.

CAVEATS: Note that the status codes used in OS-9 may not be compatible with other operating systems. A wait must be executed for each child process created.

SEE ALSO: The F$Wait system request in the "OS-9/68000 Operating System Technical Manual" and the os9fork() function.

write() Write bytes to a Path write()

UNIX System

SYNOPSIS:

```
int write(path, buffer, count)
int path;
char *buffer;
unsigned count;
```

DESCRIPTION: write() writes bytes to a path. The path number "path" is an integer which is one of the standard path numbers 0, 1, or 2, or a path number returned from a successful call to open(), creat(), create(), or dup(). The argument "buffer" is a pointer to space with at least "count" bytes of memory from which write() will obtain the data to write to the path.

It is guaranteed that at most "count" bytes will be written. If less bytes were written, this indicates an error has occurred.

writeln() causes "output-filtering" to take place such as outputting a linefeed after a carriage return, or handling the page pause facility. writeln() will write, at most, one line of data; the end of a line is indicated by a carriage-return. writeln() is the preferred call for writing to the terminal.

write() essentially does a raw write, that is, the write is performed without translation of characters. The characters are passed to file (or device) as transmitted by the program.

write() and writeln() return the number of bytes actually written. If an error occurred, -1 is returned and the appropriate error code is placed in the global variable "errno".

CAVEATS: Notice that writeln() will stop when the carriage return is written, even if the bytecount has not been exhausted. write() will do a "raw" write to the terminal; no linefeeds after return are transmitted.

SEE ALSO: The writeln() function and the I$Write and I$WritLn system requests in the "OS-9/68000 Operating System Technical Manual".

writeln() Write Bytes to a Path **writeln()**

OS-9 System

SYNOPSIS:
```
int writeln(path, buffer, count)
int path;
char *buffer;
unsigned count;
```

DESCRIPTION: See the discussion of write() for complete details on writeln().

SEE ALSO: The write() function and the I$Write and I$WritLn system requests in the "OS-9/68000 Operating System Technical Manual".

end of chapter 4

OS-9/68000 C COMPILER USER'S MANUAL
APPENDIX A
ERROR MESSAGES

COMPILER GENERATED ERROR MESSAGES

Below is a list of the error messages (listed alphabetically) that the C compiler generates, and, if applicable, probable causes and the K & R Appendix A section number (in parenthesis) to see for more specific information.

already a local variable	Variable has already been declared at the current block level. (8.1, 9.2)
argument : <text>	Error from preprocessor. Most common cause of this error is not being able to find an include file.
argument error	Function argument declared as type struct, union or function. Pointers to such types, however, are allowed. (10.1)
argument storage	Function arguments may only be declared as storage class register. (10.1)
bad character	A character not in the C character set (probably a control char) was encountered in the source file. (2)
both must be integral	>> and << operands can not be FLOAT or DOUBLE. (7.5)

break error	The break statement is allowed only inside a while, do, for or switch block (9.8)
can't take address	& operator not allowed on a register variable. Operand must otherwise be an lvalue. (7.2)
cannot cast	Type result of cast can not be FUNCTION or ARRAY. (7.2, 8.7)
cannot evaluate size	Could not determine size from declaration or initializer. (8.6, 14.3)
cannot initialize	Storage class or type does not allow variable to be initialized. (8.6)
compiler trouble	Compiler detected something it couldn't handle. Try compiling the program again. If this error still occurs, contact Microware.
condition needed	While, do, for, switch and if statements require a condition expression. (9.3)

constant expression required	Initializer expressions for static or external variables can not reference variables. They may, however, refer to the address of a previously declared variable. This installation allows no initializer expressions unless all operands are of type INT or CHAR (8.6)
onstant overflow	Input numeric constant was too large for the implied or explicit type. (2.6, [PDP-11])
constant required	Variables are not allowed for array dimensions or cases. (8.3, 8.7, 9.7)
continue error	The continue statement is allowed only inside a while, do, or for block. (9.9)
declaration mismatch	This declaration conflicts with a previous one. This is typically caused by declaring a function to return a non-integer type after a reference has been made to the function. Depending on the declaration block's line structure, this error may be reported on the line following the erroneous declaration. (11, 11.1 11.2)
divide by zero	Divide by zero occurred when evaluating a constant expression.

? expected	? is any character that was expected to appear here. Missing semicolons or braces cause this error.
expression missing	An expression is required here.
function header missing	Statement or expression encountered outside a function. Typically caused by mismatched braces. (10.1)
function type error	A function can not be declared as returning an array, function, struct or union. (8.4, 10.1)
function unfinished	End-of-file encountered before the end of function definition. (10.1)
identifier missing	Identifier name required here but none was found.
illegal declaration	Declarations are allowed only at the beginning of a block. (9.2)
label required	Label name required on goto statement. (9.11)

label undefined	Goto to label not defined in the current function. (9.12)
lvalue required	Left side of assignment must be able to be "stored into". Array names, structs functions, etc. are not lvalues. (7.1)
multiple defaults	Only one default statement is allowed in a switch block. (9.7)
multiple definition	Identifier name was declared more than once in the same block level. (9.2, 11.1)
must be integral	Type of object required here must be type int, char or pointer.
name clash	Struct-union member and tag names must be mutually distinct. (8.5)
name in a cast	Identifier name found in a cast. Only types are allowed. (7.2, 8.7)
named twice	Names in a function parameter list may appear only once. (10.1)

no 'if' for 'else'	Else statement found with no matching if. This is typically caused by extra or missing braces and/or semicolons. (9.3)
no switch statement	Case statements can only appear within a switch block. (9.7)
not a function	Primary in expression is not type "function returning...". If this is really a function call, the function name was declared differently elsewhere (7.1)
not an argument	Name does not appear in the function parameter list. (10.1)
operand expected	Unary operators require one operand, binary operators two. This is typically caused by misplaced parenthesis, casts or operators. (7.1)

out of memory	Compiler dynamic memory overflow. The compiler requires dynamic memory for symbol table entries, block level declarations and code generation. Three major factors affect this memory usage. Permanent declarations (those appearing on the outer block level (used. in include files)) must be reserved from the dynamic memory for the duration of the compilation of the file. Each { causes the compiler to perform a block level recursion which may involve "pushing down" previous declarations which consume memory. Auto class initializers require saving expression trees until past the declarations which may be very memory-expensive if many exist. Avoiding excessive declarations, both permanent and inside compound statement blocks conserve memory. If this error occurs on an auto initializer, try initializing the value in the code body.
pointer mismatch	Pointers refer to different types. Use a cast if required. (7.1)
pointer or integer required	A pointer (of any type) or integer is required to the left of the '->' operator. (7.1)
pointer required	Pointer operand required with unary * operator. (7.1)
primary expected	Primary expression required here. (7.1)

should be NULL	Second and third expression of ?: conditional operator can not be pointers to different types. If both are pointers, they must be of the same type or one of the two must be null. (7.13)
**** STACK OVERFLOW ****	Compiler stack has overflowed. Most likely cause is very deep lock-level nesting or hundreds of switch cases.
storage error	Reg and auto storage classes may only be used within functions. (8.1)
struct member mismatch	Identical member names in two different structures must have the same type and offset in both. (8.5)
struct member required	Identifier used with . and -> operators must be a structure member name. (7.1)
struct syntax	Brace, comma, etc. is missing in a struct declaration. (8.5)
struct or union inappropriate	Struct or union can not be used in this context.

syntax error	Expression, declaration or statement is incorrectly formed.
third expression missing	? must be followed by a : with expression. This error may be caused by unmatched parenthesis or other errors in the expression. (7.13)
too long	Too many characters provided in a string initializing a character array. (8.6)
too many brackets	Unmatched or unexpected brackets encountered processing an initializer. (8.6)
too many elements	More data items supplied for aggregate level in initializer than members of the aggregate. (8.6)
type error	Compiler type matching error. Should never happen.
type mismatch	Types and/or operators in expression do not correspond. (6)

typedef - not a variable	Typedef type name cannot be used in this manner. (8.8)
undeclared variable	no declaration exists at any block level for this identifier.
undefined structure	Union or struct declaration refers to an undefined structure name. (8.5)
unions not allowed	Can not initialize union members. (8.6)
unterminated character constant	Unmatched ' (character delimiters). (2.4.3)
unterminated string	Unmatched " (string delimiters). (2.5)
while expected	No while found for do statement. (9.5)

end of appendix a

OS-9/68000 C COMPILER USER'S MANUAL
APPENDIX B
COMMAND LINES / OPTIONS

COMPILER PHASE COMMAND LINES

This appendix describes the command lines and options for the individual compiler phases. Each phase of the compiler may be executed separately. The following information describes the options available to each phase.

cc (C Language Executive)

SYNTAX: cc [options] <file> {<file>} [options]
Options are not case significant.

OPTION	DESCRIPTION
-a	suppresses the assembly phase, leaving the output as assembler code in a file with the ".a" suffix.
-c	will output the source code as comments with the assembler code. This option is most useful with the -a option.
-d<identifier>	is equivalent to a "#define <identifier>" written in the source file. This option is useful where different versions of a program are maintained in one source file and differentiated through the "#ifdef" or "#ifndef" preprocessor directives. If the <identifier> is used as a macro for expansion by the preprocessor, "1"(one) will be the expanded "value" unless the form "-d<identifier>=<string>" is used in which case the expansion will be the specified <string>.
-e=<number>	sets the edition number constant byte to the specified number. This is an OS-9 convention for memory modules.
-f=<path>	overrides the output file naming conventions. The output file will be left with <path> as its name. This flag does not make sense in multiple source file mode when either the -a or -r flag is also present. The module will be called the last name in <path>.
-g	causes the linker to output a symbol module for use by the symbolic assembly language level debugger. The symbol module is placed in the execution directory using the name of the output module with the suffix: ".stb".

cc (C Language Executive)

OPTION	DESCRIPTION
-h	links the program with the "math2.l" library. To use the "math2" trap handler, include the "-x" option.
-i	links the program with the "cio.l" library which causes references to selected C I/O functions to be handled by a trap handler module called "cio".
-j	suppresses generation of a jumptable.
-l=<dir>	specifies a library file to be searched by the linker before the standard library, math libraries, and system interface library.
-m=<mem size>	instructs the linker to allocate <mem size> for the program stack. Memory size is given in kilobytes. The default stack size is approximately 2k bytes.
-o	inhibits the assembly code optimizer pass. The optimizer will shorten object code by about 11% with a comparable increase in speed. It is recommended for production versions of debugged programs.
-q	(quiet mode): the executive will not announce internal steps as they occur. Only error messages, if any, are displayed.
-r[=<dir>]	suppresses linking library modules into an executable program. Outputs are left in files with suffixes of ".r". If -r=<dir> is given, the ".r" files are left in <dir>.
-s	stops the generation of stack-checking code. This option should only be used with great care when the application is extremely time critical and when the use of the stack by compiler generated code is fully understood.
-t=<dir>	causes the executive to place the temporary files used by any compiler phase in the directory named <dir>. If the device containing the directory is the ramdisk device (e.g. -t=/r0), compilation time will be drastically reduced.

cc (C Language Executive)

OPTION	DESCRIPTION
-u<name>	will "UN-define" previously defined preprocessor macro names. Macro names pre-defined in the preprocessor are "OSK" and "mc68000". These names are useful for identifying the compiler under which the program is being compiled for purposes of writing machine and operating system independent programs.
-v=<dir>	specifies a directory to search for preprocessor "#include" files. The directory must contain ALL the "#include" files used. This path will be used instead of the DEFS directory on the system default device.
-x	causes the compiler to generate trap instructions to access the floating point math routines. This option should appear on the command line when the program is both compiled and linked (if performed separately). The linker will cause the object program to use the trap handler modules rather than extracting the code from the math libraries.

cpp (C macro preprocessor)

SYNTAX: cpp [options] <path>

<path> is read as input. cpp causes c68 to generate psect directive with last element of pathlist and _c as the psect name. If <path> is /d0/myprog.c, psect name is myprog_c. Output is always to stdout. Options are case significant.

OPTION	DESCRIPTION
-a	Raw output. No compiler control directives are produced. This is useful for stand-alone macro preprocessing.
-D<identifier>	is equivalent to a "#define <identifier>". If the <identifier> is used as a macro for expansion, "1" (one) will be the expanded "value" unless the form "-D<identifier>=<string>" is used in which case the expansion will be the specified <string>.

cpp (C macro preprocessor)

OPTION	DESCRIPTION
-E=<n> -e=<n>	Use <n> as psect edition number.
-g	Symbolic debugging flag. (Not supported).
-l	Causes c68 to copy source lines to assembly output as comments.
-o	Avoids redirection from cc68
-v=<dir>	Specifies directory containing header files for #include < > statement.

c68 (C Language Compiler)

SYNTAX: c68 [options] [<file>] [options]

If <file> is not present, c68 will read stdin. Input text need not be cpp output, but no preprocessor directives are recognized (#include, #define, macros etc.). Output assembly code is normally to stdout. Error message output is always written to stdout. To operate the compiler in "interactive mode", type "c68" on the terminal. The compiler will read standard input (looking for C source lines) and output the assembly language output to the terminal.

OPTION	DESCRIPTION
-g	Symbolic debugging flag. (Not supported).
-o=<path>	Write assembly output to <path>.
-p	Generate profile code.
-s	Suppress generation of stack checking code.
-t	Generate bsrs to trap float library instead of traps to float trap handler.

o68 (Assembly code improver)

SYNTAX: o68 [<inpath>] [<outpath>]

o68 reads stdin and writes stdout. <inpath> must be present if <outpath> is given. Since o68 rearranges and changes code, comments and assembler directives may be rearranged. o68 will recognize the -g option indicating symbolic debugging in effect. This option is for future use.

Consult the OS-9/68000 Macro Assembler User's Manual for details on the options and operation of the assembler (r68) and linker (l68).

end of appendix b

USER NOTES

OS-9/68000 C COMPILER USER'S MANUAL
APPENDIX C
FUNCTION INDEX

_cmpnam()	Compare two names	OS-9 System	4-6
_errmsg()	Print an Error Message	Miscellaneous	4-7
_exit()	Task Termination	UNIX System	4-8
_gs_devn()	Get Device Name	OS-9 System	4-9
_gs_eof()	Check for End-of-File	OS-9 System	4-10
_gs_gfd()	Get File Descriptor Sector	OS-9 System	4-11
_gs_opt()	Get Path Options	OS-9 System	4-12
_gs_pos()	Get Current File Position	OS-9 System	4-13
_gs_rdy()	Test for Data Available	OS-9 System	4-14
_gs_size()	Get Current File Size	OS-9 System	4-15
_julian()	Convert Date/Time to Julian Value ...	OS-9 System	4-16
_prgname()	Get Module Name	Miscellaneous	4-18
_s[illegible]mem()	System Memory Request	Memory Management	4-19
_srtmem()	System Memory Return	Memory Management	4-20
_ss_attr()	Set File Attributes	OS-9 System	4-21
_ss_lock()	Lock Out a Record	OS-9 System	4-22
_ss_opt()	Set Path Options	OS-9 System	4-23
_ss_pfd()	Put File Descriptor Sector	OS-9 System	4-24
_ss_rel()	Release Device	OS-9 System	4-25
_ss_rest()	Restore Device	OS-9 System	4-26
_ss_size()	Set Current File Size	OS-9 System	4-27
_ss_ssig()	Send Signal on Data Ready	OS-9 System	4-28
_ss_tiks()	Wait for Record Release	OS-9 System	4-29
_ss_wtrk()	Write Track	OS-9 System	4-30
_strass()	Structure Assignment	Miscellaneous	4-31
_sysdate()	Get Current System Date/Time	OS-9 System	4-32
_tolower()	Convert Character to Lower Case .	Character Conv.	4-34
_toupper()	Convert Character to Upper Case .	Character Conv.	4-35
abs()	Integer Absolute Value	Mathematical	4-36
access()	Determine Accessibility of a File ...	UNIX System	4-37
ac[illegible]s()	Arc Cosine	Mathematical	4-38
asin()	Arc Sine	Mathematical	4-39
atan()	Arc Tangent	Mathematical	4-40
atof()	Alpha to Floating Conversion	Character Conv.	4-41
atoi()	Alpha to Integer Conversion	Character Conv.	4-42
atol()	Alpha to Long Conversion	Character Conv.	4-43
attach()	Attach to a Device	OS-9 System	4-44
calloc()	Allocate Storage for Array	Memory Management	4-45
ceil()	Ceiling Function	Mathematical	4-46
chain()	Load and Execute a New Module	OS-9 System	4-47
chainc()	Load and Execute a New Module	OS-9 System	4-47
chdir()	Change Current Data Directory	UNIX System	4-49
chmod()	Change File Access Permissions	UNIX System	4-50
chown()	Change Owner of a File	UNIX System	4-51
chxdir()	Change Current Execution Directory ..	OS-9 System	4-52
cleareof()	Clear End of File Condition	Standard I/O	4-53
clearerr()	Clear Error Condition	Standard I/O	4-54
close()	Close a Path	UNIX System	4-55
closedir()	Close the Named Directory Stream ..	Miscellaneous	4-56
cos()	Cosine	Mathematical	4-57

OS-9/68000 C COMPILER USER'S MANUAL
APPENDIX C
FUNCTION INDEX

crc()	Calculate Module CRC	OS-9 System	4-58
creat()	Create a File	UNIX System	4-60
create()	Create a File	OS-9 System	4-61
detach()	Detach a Device	OS-9 System	4-62
dup()	Duplicate a Path	UNIX System	4-63
ebrk()	Obtain External Memory	Memory Management	4-64
exit()	Terminate Task	UNIX System	4-65
exp()	Exponential Function	Mathematical	4-66
fabs()	Floating Absolute Value	Mathematical	4-67
fclose()	Close a File........................	Standard I/O	4-68
fdopen()	Attach a Path to a File Pointer	Standard I/O	4-69
feof()	Check Buffered File for End of File	Standard I/O	4-70
ferror()	Check Buffered File for Error Condition	Standard I/O	4-71
flush()	Flush a File's Buffer	Standard I/O	4-72
fgetc()	Get a Character from a File	Standard I/O	4-73
fgets()	Get a String from a File	Standard I/O	4-74
fileno()	Determine Path Number from File	Standard I/O	4-75
findnstr()	Search String for Pattern	String Handling	4-76
findstr()	Search String for Pattern	String Handling	4-77
floor()	Floor Function	Mathematical	4-78
fopen()	Open a File	Standard I/O	4-79
fprintf()	Formatted Output	Standard I/O	4-81
fputs()	Output a String to a File	Standard I/O	4-82
fread()	Read Data from a File	Standard I/O	4-83
free()	Return Memory to Arena	Memory Management	4-84
freemem()	Determine size of Unused Stack Area	Miscellaneous	4-85
freopen()	Re-Open a File	Standard I/O	4-86
fscanf()	Input String Conversion	Standard I/O	4-87
fseek()	Reposition File Pointer	Standard I/O	4-88
tell()	Report File Pointer Position	Standard I/O	4-89
fwrite()	Write Data to a File	Standard I/O	4-90
getc()	Get Next Character from File	Standard I/O	4-91
getchar()	Get Next Character from stdin	Standard I/O	4-93
getenv()	Get Value for Environment Name	UNIX System	4-94
getime()	Get System Time	UNIX System	4-95
getpid()	Determine Process ID Number	UNIX System	4-96
gets()	Get a String from a File	Standard I/O	4-97
getstat()	Get File Status	OS-9 System	4-98
getuid()	Determine User ID Number	UNIX System	4-100
getw()	Read a Word from a File	Standard I/O	4-101
hypot()	Euclidean Distance Function	Mathematical	4-102
ibrk()	Request Internal Memory	Memory Management	4-103
index()	Search for Character in String ..	String Handling	4-104
intercept()	Set up Process Signal Handler	OS-9 System	4-105
isalnum()	See if Argument is Alphanumeric	Char. Class.	4-107
isalpha()	See if Argument is Alphabetic	Char. Class.	4-108
isascii()	See if Argument is ASCII	Char. Class.	4-109
iscntrl()	See if Argument is a Control Code ..	Char. Class.	4-110

OS-9/68000 C COMPILER USER'S MANUAL
APPENDIX C
FUNCTION INDEX

isdigit()	See if Argument is a Digit	Char. Class.	4-111
islower()	See if Argument is Lower Case	Char. Class.	4-112
isprint()	See if Argument is a Printable Character	Char. Class.	4-113
ispunct()	See if Argument is a Punctuation Character	Char. Class.	4-114
isspace()	See if Argument is White Space	Char. Class.	4-115
isupper()	See if Argument is Upper Case	Char. Class.	4-116
isxdigit()	See if Argument is a Hex Character .	Char. Class.	4-117
kill()	Send a Signal to a Process	UNIX System	4-118
log()	Natural Logarithm	Mathematical	4-119
log10()	Natural Logarithm Base Ten	Mathematical	4-120
lo[illegible]jmp()	Non-local Goto	Miscellaneous	4-121
lseek()	Position File Pointer	UNIX System	4-123
malloc()	Allocate Memory from an Arena .	Memory Management	4-124
mknod()	Create a Directory	UNIX System	4-125
mktemp()	Create a File with a Unique Name ..	Miscellaneous	4-126
modf()	Return Parts of a Real Number	Mathematical	4-127
modlink()	Link to a Memory Module	OS-9 System	4-128
modload()	Load and Link to a Memory Module	OS-9 System	4-129
munlink()	Unlink from a Module	OS-9 System	4-130
munload()	Unload a Module	OS-9 System	4-131
open()	Open a File	UNIX System	4-132
opendir()	Open a Directory	Miscellaneous	4-133
os9exec()	OS-9 System Call Processing	OS-9 System	4-134
os9fork()	Create a Process	OS-9 System	4-136
os9forkc()	Create a Process	OS-9 System	4-136
pause()	Wait for Signal	UNIX System	4-137
pffinit()	Initialize for Float Output (obsolete)	Standard I/O	4-138
pf[illegible]nit()	Initialize for Longs Output (obsolete)	Standard I/O	4-139
pow()	Power Function	Mathematical	4-140
prerr()	Print error Message	UNIX System	4-141
printf()	Formatted Output	Standard I/O	4-142
putc()	Put Next Character to File	Standard I/O	4-145
putchar()	Put Next Character to Standard Output	Standard I/O	4-147
puts()	Output a String to a File	Standard I/O	4-148
putw()	Put a Word to a File	Standard I/O	4-149
qsort()	Quick Sort	Miscellaneous	4-150
read()	Read bytes from a Path	UNIX System	4-151
readdir()	Returns Pointer to Next Directory Entry	Miscellaneous	4-152
readln()	Read bytes from a Path	OS-9 System	4-153
rewind()	Return File Pointer to Zero	Standard I/O	4-154
rewinddir()	Reset position of the Directory Stream	Miscellaneous	4-155
rindex()	Search for Character in String ..	String Handling	4-156
sbrk()	Extend Data Memory Segment	Memory Management	4-157

OS-9/68000 C COMPILER USER'S MANUAL
APPENDIX C
FUNCTION INDEX

scanf()	Input Strings Conversion	Standard I/O	4-158
seekdir()	Sets the Position of the Next Readdir()	Miscellaneous	4-161
setbuf()	Fix File Buffer	Standard I/O	4-162
setime()	Set System Time	UNIX System	4-163
setjmp()	Non-local Goto	Miscellaneous	4-164
setpr()	Set Process Priority	UNIX System	4-165
setstat()	Set File Status	OS-9 System	4-166
setuid()	Set User ID	UNIX System	4-167
sin()	Sine Function	Mathematical	4-168
sleep()	Suspend Execution for a Time	UNIX System	4-169
sprintf()	Formatted Output	Standard I/O	4-170
sqrt()	Square Root Function	Mathematical	4-171
sscanf()	Input Strings Conversion	Standard I/O	4-172
tacksiz()	Obtain Size of Stack Used	Miscellaneous	4-173
strcat()	String Catenation	String Handling	4-174
strcmp()	String Comparison	String Handling	4-175
strcpy()	String Copy	String Handling	4-176
strhcpy()	Copy Old OS-9 Strings	String Handling	4-177
strlen()	Determine String Length	String Handling	4-178
strncat()	String Catenation	String Handling	4-179
strncmp()	String Comparison	String Handling	4-180
strncpy()	String Copy	String Handling	4-181
system()	Shell Command Execution	Miscellaneous	4-182
tan()	Tangent Function	Mathematical	4-183
telldir()	Returns the Current Location	Miscellaneous	4-184
toascii()	Character Translation	Character Conv.	4-185
tolower()	Character Translation	Character Conv.	4-186
toupper()	Character Translation	Character Conv.	4-187
tsleep()	Sleep for Specified Interval	OS-9 System	4-188
ungetc()	Unget a Character	Standard I/O	4-189
nlink()	Unlink (Delete) a File	UNIX System	4-190
nlinkx()	Unlink (Delete) a File	OS-9 System	4-191
wait()	Wait for Process Termination	UNIX System	4-192
write()	Wtie bytes to a Path	UNIX System	4-193
writeln()	Write Bytes to a Path	OS-9 System	4-194

end of appendix c

OS-9/68000 C COMPILER USER'S MANUAL

INDEX

For an alphabetical index of the Standard Library Functions, see Appendix C.

-------------- A --------------

Argument passing 3-4, 3-5
Assembly Language
  Imbedded code 2-4
  Interface 3-3

-------------- C --------------

C executive (cc) 1-2, through 1-8, 3-1, B-1, B-2, B-3, B-4
  Command line
    Examples 1-7, 1-8,
    Execution 1-2, 1-3, 1-4
    Options 1-5, 1-6, 1-7, B-1, B-2, B-3, B-4
    Parameters 2-3
    Syntax 1-2, B-1
Clib.l 1-1
Clibn.l 1-1
Compiler
  Command line
    Options B-4
    Syntax B-4
  Cstart 3-2
  Executable files 1-1
  Library files 1-1
  Organization 3-1
  Stack 3-3
Cstart 1-1, 3-2
Cross compiler versions i

-------------- D --------------

Data types 2-1
  double precision format 2-2
  floating point format 2-2
DDEV 1-2
Default System Device 1-2

-------------- E --------------

End-of-line character 2-3
Errors
  Handling 3-1, 3-2
  Messages Apdx A
Escape sequences 2-6

-------------- I --------------

Indexed register indirect with offset addressing mode 2-4

-------------- J --------------

Jumptable 3-5, 3-6

-------------- M --------------

Math.l 1-1

-------------- P --------------

Preprocessor (ccp) 3-1, B-3
  Command line
    Options B-3
    Syntax B-3

-------------- O --------------

Object code improver (o68) 3-1

-------------- R --------------

Registers
  Allocation 3-4
  Argument passing 3-4, 3-5
  Variables 2-1
Remote Storage Class 2-4, 2-5 2-6
  Declaration Syntax 2-6

-------------- S --------------

Stack 3-3
Standard Library 2-7, Chap 4
  Function Index Apdx C
Sys.l 1-1